Simulateur d'activités de la vie quotidienne et validation de scénario

par

Pierre Busnel

mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, avril 2008

Your file Votre référence
ISBN: 978-0-494-49469-1
Our file Notre référence
ISBN: 978-0-494-49469-1

Le 22 avril 2008

le jury a accepté le mémoire de M. Pierre Busnel dans sa version finale.

Membres du jury

Mme Hélène Pigot
Directrice
Département d'informatique

M. Sylvain Giroux
Membre
Département d'informatique

M. Richard St-Denis
Président-rapporteur
Département d'informatique

Sommaire

La simulation des activités de la vie quotidienne (AVQ) dans le contexte des habitats intelligents a pour but l'obtention rapide et massive de données similaires aux sorties des capteurs qui composent l'habitat intelligent. Rapide, pour ne pas avoir à observer une même activité pendant plusieurs mois, massive pour obtenir une base de connaissances substantielle des activités à observer. Pour atteindre cet objectif, les concepts relatifs aux AVQ tels que la composition des activités, les ressources et les paramètres que l'on souhaite observer doivent être extraits du monde réel pour être transposés dans le monde de la simulation.

Le processus de développement du simulateur d'AVQ prend l'observation du monde réel comme point de départ et décrit les étapes qui amènent à l'implémentation puis à la validation du simulateur. Le simulateur, ayant déjà fait l'objet d'un travail de maîtrise, a été refactorisé pour dégager les différents concepts relatifs aux AVQ avant d'être validé. Dans ce contexte, nous abordons d'une part, la méthodologie liée à la refactorisation du code original du simulateur pour refléter plus précisément le monde réel et faciliter l'ajout de fonctionnalités nécessaires au bon déroulement de la simulation. D'autre part, nous étudions les techniques de validation existantes et proposons celles à implémenter pour valider le simulateur d'AVQ. Nous implantons deux techniques, le *data historical validation* et la validation par animation, que nous testons sur un scénario observé dans le milieu réel.

Remerciements

J'adresse mes sincères remerciements aux personnes suivantes pour leur aide et leur soutien tout au long de mes études de maîtrise.

Hélène Pigot, ma directrice de recherche, pour son soutien, ses conseils, sa confiance, ainsi que pour m'avoir ouvert au monde de l'*informatique au service des Hommes.* Je tiens également à remercier Sylvain Giroux pour ses précieux conseils en programmation, en refactorisation. A Hélène et Sylvain, je vous remercie aussi pour l'expérience que vous m'avez permis d'acquérir ces dernières années dans le monde de la recherche.

Merci à l'ensemble du laboratoire DOMUS pour les conseils, la bonne humeur, les encouragements et pour tout ce qui fait que l'on est fier de faire partie de la même équipe. Merci notamment à Francis Bouchard pour son expertise, son aide précieuse et ses expressions qui ont enrichi mon vocabulaire québécois.

Abdelkader Ghouraf pour m'avoir ouvert le chemin sur la simulation d'AVQ, Nadine Gagnon et Mickaël Beaudry pour leur travail remarquable sur le prototype *lo-fi* et Pierre-Yves Groussard pour ses tomates-Mozarella.

A ma famille et à mes amis pour être là malgré la distance, et pour m'avoir soutenu toutes ces années. Notamment ma mère, pour ses encouragements et son soutien, Bruno Dufour pour ses conseils qui m'ont fort aidé dans mes choix d'études, et François Courtemanche pour m'avoir montré le vrai visage du Québec.

Table des matières

Liste des tableaux

Liste des figures

Introduction

Le vieillissement de la population

Les pays occidentaux font face à une augmentation de la population vieillissante, conjointement à une baisse du taux de natalité. D'après une estimation du bureau de recensement américain, le nombre de personnes âgées de 65 ans et plus, était de 36 millions en 2003 et devrait atteindre 72 millions en 2030 et 86.7 millions en 2050 [7]. On retrouve des chiffres proportionnellement similaires dans l'ensemble des pays occidentaux. Cette situation entraîne une augmentation de maladies ou d'accidents dus au vieillissement normal ou pathologique qui conduit à la perte d'autonomie et réduit la qualité de vie des personnes âgées.

La maladie d'Alzheimer est l'une des maladies les plus connues liées à l'âge. Selon le sondage précédemment cité, on estimait à 4,5 millions le nombre d'américains atteints de la maladie d'Alzheimer en 2000 et que ce nombre atteindrait 13,2 millions en 2050 [6].

La maladie d'Alzheimer est une maladie dégénérative qui provoque principalement des pertes cognitives [4]. Des déficits de mémoire, de planification et de jugement entraînent une perte d'autonomie. Dans les premiers temps de la maladie, les malades ressentent des difficultés lors de la réalisation de tâches inhabituelles ou complexes, mais restent toutefois capables de vivre dans un environnement familier et d'effectuer des tâches simples. Les risques d'isolement, de malnutrition, de brûlures, d'inondations et d'incendies augmentent au fur et à mesure de l'évolution de la maladie. Dans les derniers moments de la maladie,

les malades nécessitent une assistance continue, qui entraîne fréquemment un transfert en milieu hospitalier.

Malgré ces handicaps, 92% des personnes âgées souhaitent rester chez elles et 40% préfèrent passer les derniers moments de leur vie dans leur maison [7]. Les personnes agées dépendent de leur famille et d'aides-soignants pour les aider dans leur vie quotidienne, ce qui devient rapidement très difficile pour les aidants [4]. Afin d'alléger le fardeau des proches, un environnement intelligent pourrait compenser les pertes cognitives des personnes agées en leur fournissant des indices pour réaliser plus facilement leurs activités de la vie quotidienne (AVQ).

L'assistance cognitive

Un habitat intelligent interagit avec l'occupant pour améliorer ou lui redonner son autonomie grâce à l'informatique diffuse [12]. Contrairement au concept de télé-assistance, où l'aidant se situe à l'extérieur, les habitats intelligents sont similaires à un système clos, composé de l'habitat et de son occupant. L'habitat intelligent est généralement équipé de différents types de capteurs connectés à un serveur qui emmagasine et analyse les données collectées grâce à des techniques d'intelligence artificielle. L'habitat intelligent peut ensuite intervenir pour éviter des situations à risque et pour aider l'occupant à terminer ses activités [17].

L'assistance peut se diviser en deux parties : l'assistance physique et l'assistance cognitive. Dans la première, l'environnement vise à compenser le handicap physique en proposant des outils appropriés tels que des télécommandes, la reconnaissance vocale, une chaise roulante, etc. Dans le cas de l'assistance cognitive, l'environnement vise à compenser les pertes cognitives de l'individu en lui donnant des indices et en interagissant avec lui selon les besoins de la situation.

Le laboratoire Domus

Au laboratoire Domus de l'Université de Sherbrooke, l'assistance cognitive s'oriente vers l'aide aux personnes souffrant de troubles cognitifs pour y développer des outils d'assistance cognitive [5]. La création de ces outils soulève des problèmes en matière d'informatique diffuse et contextuelle, d'intelligence artificielle, de réseautique et de conception d'interface [12].

Le laboratoire Domus comprend un appartement fonctionnel, parsemé de capteurs électromagnétiques afin de détecter l'ouverture des portes, de capteurs de mouvement ainsi que de tapis tactiles pour localiser l'occupant. L'activité réalisée par l'occupant au sein de l'appartement est reconnue grâce à l'analyse des données issues des capteurs.

Dans la vie quotidienne, nous réalisons des activités de manières différentes. Par exemple, lors de la routine matinale, l'un peut se lever à un moment différent selon le jour de la semaine, aller à la salle de bain puis prendre son petit-déjeuner. Il peut également être interrompu par un appel téléphonique ou décider de rester plus longtemps dans la cuisine pour écouter la radio. Il y a donc une multitude de déroulements possibles pour la même activité. Les capteurs sont alors activés dans un ordre différent, à un moment différent, et pour une durée différente. Afin d'améliorer la reconnaissance d'activités, les données de différents scénarios d'une même activité doivent être disponibles pour réaliser un apprentissage de ces scénarios. Malheureusement, obtenir de telles données prendrait des mois voire des années d'enregistrement. Un simulateur d'AVQ est nécessaire pour générer des sorties capteurs similaires à celles issues de la vie quotidienne où une activité peut être interrompue ou s'étendre sur une durée variable.

Le simulateur d'AVQ

Abdelkader Ghouraf présente dans son mémoire les fondements théoriques d'un simulateur d'AVQ, par exemple les réseaux de Petri, les interruptions, les ressources de l'individu et de l'environnement [10], [2]. Cette étude a abouti au développement d'un prototype du simulateur implémentant en partie ces fondements. Cependant, l'utilisation du simulateur pour produire des agendas d'activités proches de la réalité nécessite davantage de développement. Les sorties capteurs produites par le simulateur doivent donc être validées afin d'en assurer la cohérence avec les données de la vie réelle.

Le chapitre 1 présentera une revue de littérature axée sur la théorie de la simulation, les composants de la vie réelle et ceux du monde de la simulation, pour en expliquer les principaux fondements. Le chapitre 2 établira les objectifs à atteindre en matière de maintenance du code d'une part, de validation des sorties du simulateur d'autre part. La méthodologie entreprise pour atteindre ces objectifs sera ensuite détaillée. Les résultats obtenus, tant au niveau du code qu'au niveau de la validation, seront illustrés dans le chapitre 3 avant d'être discuter dans la conclusion.

Chapitre 1

Revue de littérature

Depuis plusieurs années, la simulation fait l'objet de recherches scientifiques. Les protagonistes de ces recherches tentent de standardiser la modélisation de systèmes complexes. Dans ce chapitre, nous parcourons les différentes étapes de modélisation, établissons les objectifs à atteindre pour chaque étape ainsi que les moyens employés pour y parvenir. Ces objectifs concernent d'une part le code du simulateur, c'est-à-dire sa structure et sa lisibilité, et la validation des sorties générées par le simulateur d'autre part.

1.1 Théorie de la simulation

Depuis plusieurs dizaines années, des travaux de recherche sont effectués pour modéliser rapidement et efficacement des systèmes complexes [18], [1], [13]. Le but est de mettre au point des simulateurs capables de produire des résultats similaires à ceux obtenus dans la réalité mais en plus grande quantité et dans un laps de temps plus court. Le modèle de simulation doit donc refléter le plus fidèlement possible la réalité. Une observation fine et une analyse profonde du monde réel sont nécessaires lors du processus de développement. Associés à la perception du monde réel, les objectifs du simulateur

doivent définir la problématique à résoudre, afin d'élaborer des théories du système à observer.

Sargent et al. proposent un cycle de développement présentant les différentes étapes à suivre lors du développement d'un simulateur et l'étroite relation entre le monde réel et le monde de la simulation [18], [1]. La figure 1 illustre le paradigme de Sargent. Celui-ci part de l'observation du monde réel et des données collectées dans ce monde pour formuler des théories qui sont, par la suite, exprimées dans le monde de la simulation. Les deux prochaines sections présentent le monde réel et le monde de la simulation.

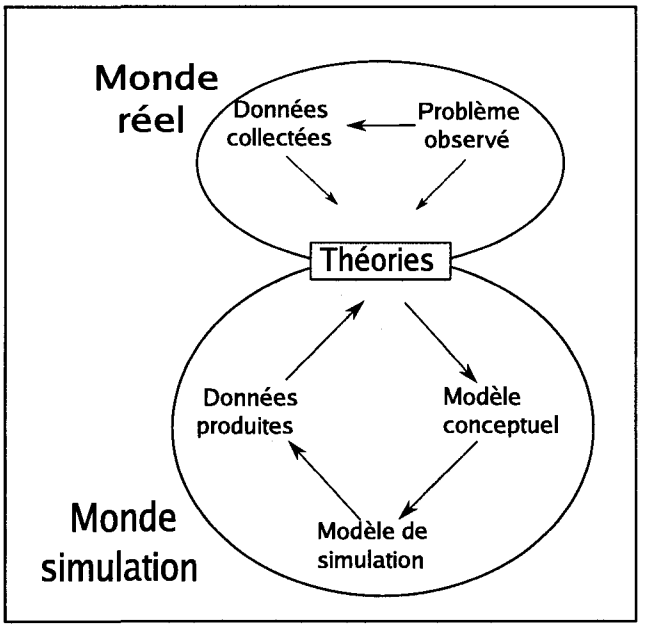

Figure 1 – Cycle de développement d'un modèle de simulation d'après Sargent

1.2 La réalisation des AVQ dans le monde réel

Dans ce contexte, le monde réel est composé d'un individu réalisant des AVQ dans un habitat comportant des capteurs. Durant la journée, cet individu prépare son dîner, fait sa toilette, regarde la télévision et va se coucher. Parfois ces activités sont réalisées sans interruption, parfois certaines sont réalisées en parallèle ou sont interrompues par une activité prioritaire comme répondre au téléphone pendant la préparation du dîner. Une journée est divisée en tranches horaires pendant lesquelles certaines activités ont plus de probabilités d'être exécutées. Ces probabilités dépendent des habitudes de vie de l'individu observé. Les capteurs répartis dans l'environnement collectent des données selon les activités réalisées. Les théories décrivent les caractéristiques suivantes du monde réel : les AVQ et les habitudes de vie, la gestion des interruptions, l'environnement et ses capteurs, les ressources liées à l'environnement et celles liées à l'individu.

1.2.1 Les AVQ et les habitudes de vie

Selon le guide canadien de l'ergothérapie, une AVQ est une "activité réalisée dans le but d'assouvir un besoin d'un individu dans le but de prendre soin de sa personne et de sa propriété" [3]. L'AVQ se décompose hiérarchiquement en tâches et sous-tâches jusqu'à atteindre la forme atomique d'une action perceptible par les capteurs. Par exemple, l'AVQ *faire le café* est composée de tâches, telles que *faire bouillir de l'eau*, et d'actions, telles que *ouvrir le robinet*. Une action altère l'état de l'environnement et de la personne. Après avoir bu un verre d'eau, l'individu n'a plus soif et le verre est vide. Les actions modifient également les états des tâches et sous-tâches dont elles font partie comme décrit dans le cycle de vie des activités. Les habitudes de vies représentent les routines qui sont exécutées régulièrement comme boire du café six jours par semaine entre 7h30 et 8h du matin et manger des oeufs bacon le dimanche matin. Les habitudes de vie jouent un rôle significatif pour prédire et reconnaître plus précisément les AVQ réalisées.

1.2.2 Les interruptions

Puisque nous ne pouvons prévoir un éventuel appel téléphonique ou la visite d'un ami qui frappe à la porte, nous devons adapter l'activité en cours de réalisation en cas de situation imprévisible. Dans le monde réel, une activité interrompue peut être reprise ultérieurement, recommencée voire abandonnée. Par exemple, si le téléphone sonne pendant que de l'eau est en train de bouillir, l'on peut éteindre l'élément de cuisson pour répondre en toute sécurité. Selon la durée de l'appel téléphonique, soit l'eau est toujours chaude, soit elle a besoin d'être réchauffée. Dans le premier cas, la tâche peut être reprise, dans le deuxième, il est nécessaire de recommencer la tâche depuis le début. Le temps de validité de la tâche indique la possibilité de la reprendre ou de la recommencer.

1.2.3 Le cycle de vie des activités

Les AVQ et les tâches évoluent selon un cycle de vie des activités (figure 2). Ce cycle comprend neuf états.

- Libre

  L'état *libre* représente l'état neutre d'une activité. Pour sortir de cet état, une activité doit être prévue ou devenir une interruption. Pour y retourner, une activité doit préalablement être complétée, abandonnée ou annulée.

- Prévue

  Une activité est *prévue* lorsqu'elle fait partie de la liste des activités à simuler. Dans cet état, aucune condition n'est vérifiée.

- Non prévue

  Une activité est *non prévue* lorsqu'elle fait partie de la liste des interruptions à simuler. Dans cet état, aucune condition n'est vérifiée.

- En attente

  Pour être mise *en attente*, une activité doit satisfaire ses préconditions telles que la

disponibilité des ressources qui lui sont nécessaires.

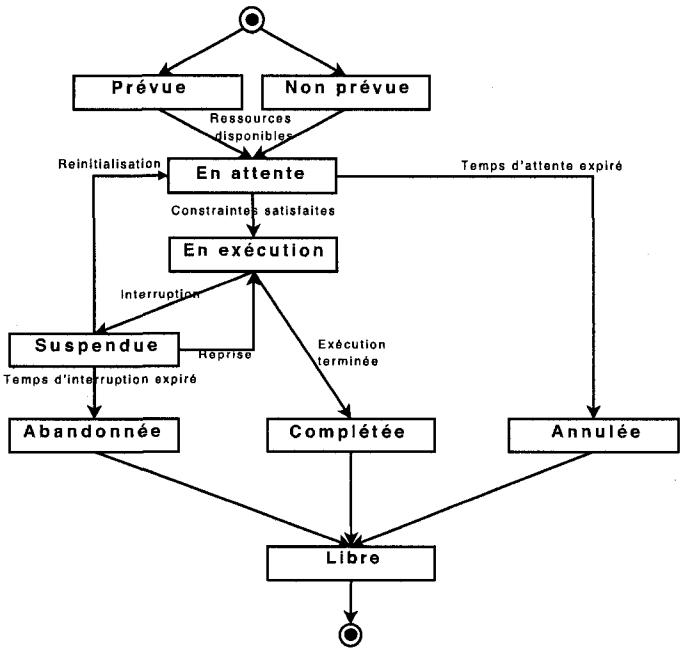

Figure 2 – Cycle de vie des activités

- En exécution

  Une activité entre *en exécution* si les préconditions sont satisfaites et si le temps d'attente n'a pas été dépassé. Parmi les préconditions, on retrouve également la liste des activités devant être *complétées* avant l'activité en cours.

- Suspendue

  Lorsqu'une interruption se produit et n'est pas ignorée, l'activité en cours est *suspendue.* Une fois l'interruption traitée, l'activité sera soit recommencée depuis le début (mise *en attente* puis *en exécution*), soit reprise (*en exécution*), soit *abandonnée.*

- Annulée

  Si une activité *en attente* n'a toujours pas commencé au-delà de son temps de

départ maximal. Son temps d'attente est dépassé et l'activité est *annulée* puis devient *libre.*

- Abandonnée

  Si une activité *suspendue* n'est pas reprise au delà de son temps de validité, elle est *abandonnée* puis devient *libre.*

- Complétée

  Lorsqu'une activité atteint son temps d'exécution, celle-ci est *complétée* avant de devenir *libre.*

1.2.4 Les ressources

Les ressources se définissent par chaque item nécessaire à la réalisation d'une activité. Ces ressources sont soit liées à la personne, soit liées à l'environnement et peuvent être limitées dans le temps. Ces ressources, essentielles au bon déroulement d'une tâche, se déclinent en trois types.

- Personnelles

  D'après la théorie VACP (visuel, audio, cognitif et psychomoteur) de Wickens, les ressources personnelles sont composées de quatre composants [19]. Ces composants représentent respectivement la capacité de voir, entendre, se souvenir des connaissances procédurales et déclaratives, et celle d'accomplir des mouvements. Via le simulateur, il est également possible d'y ajouter d'autres ressources liées à la personne telles que la voix (en cas de commande vocale ou pour répondre au téléphone), la fatigue (provoquée par une activité éprouvante) ou le stress.

- Physiques et consommables

  Ce type de ressource regroupe les substances ou produits consommables de la vie quotidienne tels que l'eau, le savon ou le lait.

- Physiques et matérielles

  Les ressources matérielles comprennent l'équipement de l'appartement tel que les

électroménagers, les meubles ou les ustensiles de cuisine.

Ces ressources soutiennent les tâches et ont une influence sur leur mise en exécution. Certaines peuvent être utilisées pour plus d'une tâche simultanément, par exemple, une cuisinière équipée de quatre éléments de cuisine peut servir à la fois pour faire bouillir de l'eau et cuire des œufs. Enfin, l'exécution d'une activité peut provoquer l'amoindrissement d'une ressource voire sa disparition.

1.2.5 L'environnement

L'environnement est défini par la configuration des pièces et la répartition des capteurs. La première est essentielle pour estimer les chemins possibles entre les pièces de l'appartement, la dernière donne des informations sur la position, l'état physiologique ou les actions de l'individu. Les capteurs utilisés sont bistables ou analogiques. Un capteur bistable est un capteur qui a deux états de repos, ces deux états étant stables. Le signal envoyé indique l'état actuel du capteur. Par exemple, les tapis tactiles, les capteurs électromagnétiques et les capteurs de mouvement sont bistables. Un capteur analogique envoie des données représentant une valeur quantifiable telle que la température de la pièce.

1.3 Le monde de la simulation des activités de la vie quotidienne

Le monde de la simulation des AVQ inclut notre perception de la réalité, exprimée par des théories à partir desquelles un modèle conceptuel est construit [18].
Cette partie présente les composants de la simulation des AVQ à partir du modèle conceptuel représenté grâce aux réseaux de Petri, jusqu'à l'implémentation et sa validation.

1.3.1 Le modèle conceptuel

Les réseaux de Petri décrivent les séquences d'activités, les ressources et les éventuelles interruptions. Leur principal intérêt réside dans la richesse d'expression qu'ils permettent grâce à un langage graphique et à la représentation formelle [14], [15]. Les caractéristiques dynamiques du système sont représentées par le réseau de Petri dans le but de comprendre son comportement et d'étudier ses performances.

Les réseaux de Petri se composent de *places*, de *transitions* et d'*arcs*. Un *arc entrant* connecte une place à une transition et un *arc sortant* connecte une transition à une place. Une place peut contenir plusieurs jetons consommables. Lorsqu'une place connectée à une transition contient le nombre de jetons requis pour *franchir une transition*, la transition est déclenchée, consomme les jetons en entrée et produit des jetons dans la place sortante. Le nombre de jetons ajoutés dans la place sortante n'est pas nécessairement égal au nombre de jetons consommés et dépend de la transition. L'état courant du système est donné par le nombre des jetons répartis dans chaque place du réseau de Petri et est appelé *marquage*.

Dans le simulateur d'AVQ, les places représentent, d'une part, les préconditions à satisfaire dans le but de franchir la transition, et d'autre part, les postconditions qui sont satisfaites après le déclenchement de la transition. Une transition est liée à une activité ; lorsqu'elle est déclenchée, une activité est exécutée.

La figure 3 utilise l'activité *faire du café instantané* comme exemple d'utilisation d'un réseau de Petri. Les cinq places (de C0 à C4) représentent les conditions reliées à l'environnement, telles qu'avoir de l'eau et avoir une casserole. Les jetons dans les places C0, C1 et C3 symbolisent les conditions associées et satisfaites. Quand des jetons sont dans les places C0 et C1, les préconditions sont satisfaites et la transition déclenche l'événement E1. Dans l'état suivant, les places C0 et C1 sont vides et C2 contient un jeton. A moins de manquer de café soluble, la place C3 contient toujours un jeton.

Cet exemple présente une activité qui est complétée sans interruption. Le cycle de

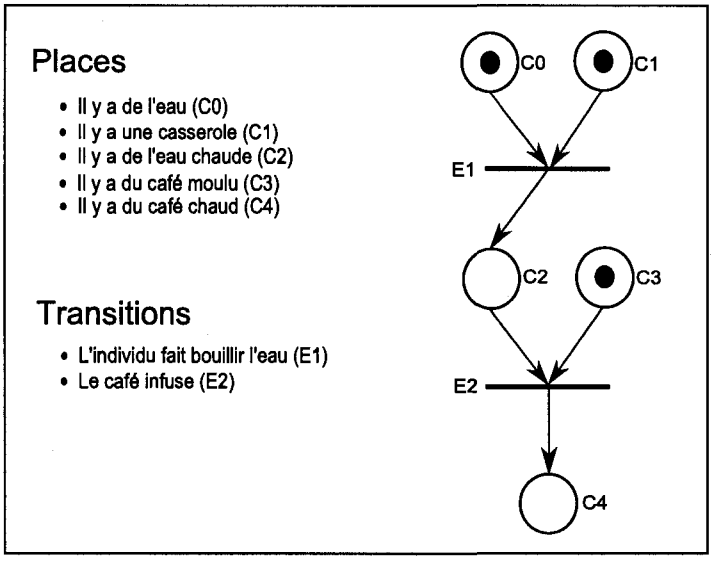

Figure 3 – Réseau de Petri modélisant l'AVQ *faire du café instantané*

vie d'une activité décrit précédemment, propose cinq chemins différents pour une tâche à travers les neuf états possibles. Ces cinq chemins, qui font partie du réseau de Petri global, sont les suivants :

- La tâche est complétée sans interruption.
- Les préconditions de la tâche ne sont jamais satisfaites. La tâche est annulée une fois le temps d'attente dépassé.
- La tâche est interrompue puis reprise jusqu'à être complétée.
- La tâche est interrompue trop longtemps, puis est réinitialisée.
- La tâche est interrompue trop longtemps, puis est abandonnée.

1.3.2 La refactorisation

Définition de la refactorisation

La refactorisation se définit comme un processus de modification du code source d'un programme de manière à améliorer sa lisibilité et sa réutilisation sans en modifier son comportement extérieur [8].

L'élaboration d'un logiciel consiste, dans un premier temps, à spécifier la structure de code, c'est-à-dire, à définir à l'avance la hiérarchie des classes et les signatures des méthodes qui composeront par la suite le code. Lorsque cette structure est établie, le développeur commence à coder. Cela correspond à l'approche traditionnelle en génie logiciel qui comprend les "procédures, méthodes, langages, ateliers, imposés ou préconisés par les normes adaptées à l'environnement d'utilisation afin de favoriser la production et la maintenance de composants logiciels de qualité" selon Jaulent [11].

La refactorisation diffère de l'approche précédente, puisqu'elle se pratique à partir de code existant. L'objectif consiste à modifier progressivement la structure du code en renommant, en déplaçant ou en extrayant des variables et des méthodes jusqu'à obtenir un code sain. Même si le programme original était codé de manière anarchique, celui-ci devient de mieux en mieux structuré au fur et à mesure de la refactorisation.

Motivations

Si l'approche traditionnelle du génie logiciel permet de concevoir un programme sain et viable avant même l'écriture du code, quel est l'intérêt de la refactorisation ?

De nombreux projets de programmation voient le jour par une demande de création de petites applications devant répondre à un besoin urgent. Dans cette optique, l'application doit être opérationnelle rapidement, sans prendre le temps de penser au cycle de vie du programme ou à son architecture. Les lignes de code qui en résultent, produisent le résultat demandé mais sont bien souvent compréhensibles seulement par celui qui les a

écrites. L'accumulation de nouvelles fonctionnalités, l'ajout et la modification successifs de code par d'autres programmeurs, sans réflexion approfondie sur l'architecture globale du code, rendront la lisibilité du programme de plus en plus difficile et risquent également d'amener de nouveaux bogues [8].

Les responsables de projets renoncent souvent à retravailler le code existant partant du principe suivant "si le programme fonctionne, il ne faut plus y toucher". Pourtant, prendre le temps de refactoriser permet d'en gagner considérablement par la suite.

Difficultés de la refactorisation

La refactorisation est un processus qui peut s'avérer à la fois difficile et risqué. Elle peut d'une part introduire de nouvelles erreurs très difficiles à détecter, mais peut également ramener de plusieurs semaines en arrière le projet de développement si elle est faite de manière informelle. Le principe même de refactorisation étant récent, les problèmes qu'elle peut engendrer ne sont pas encore tous référencés et solutionnés.

Étapes de la refactorisation

Quand devient-il nécessaire de refactoriser et comment s'y prendre ? Il n'existe pas de règle précise qui réponde à cette question, cependant, les auteurs d'ouvrages sur la refactorisation s'accordent à dire que si le code "sent mauvais", il faut le modifier. En d'autres termes, si lors de la lecture du code, celui-ci est incompréhensible, si le rôle d'une classe n'est pas clairement défini, ou si elle comporte un trop grand nombre de variables ou des méthodes trop longues, une réorganisation s'impose. Ce constat est toutefois laissé au libre arbitre du programmeur qui devra utiliser son bon sens et son expérience pour atteindre les objectifs de refactorisation souhaités.

Il existe plusieurs mécanismes de refactorisation ; Fowler en identifie plus de 70 regroupés par catégorie d'action. Parmi ceux-ci, on retrouve les mécanismes suivants.

- Composition de méthodes

Bien souvent, la logique d'une méthode est difficile à cerner parce qu'elle est trop longue. Des méthodes trop longues sont problématiques car elles contiennent trop d'informations difficiles à identifier et la maintenance du code est difficile. La composition de méthodes permet de redessiner l'aspect de certaines méthodes souvent devenues trop complexes, car trop longues.

Extract method : Extrait un bout de code pour en faire une méthode.

Replace temp with query : Sert à remplacer une variable temporaire par une invocation de méthode, toutes les références à cette variable temporaire par la nouvelle méthode.

- Déplacement de composants entre objets

  Move method : Crée une nouvelle méthode similaire dans la classe qui l'utilise le plus. L'ancienne méthode est soit supprimée, soit devient une délégation.

  Extract class : Permet à partir d'une classe, d'en créer une seconde, en bougeant les champs pertinents de l'ancienne à la nouvelle classe.

- Simplification des expressions conditionnelles

  Replace conditional with polymorphism : Dépendamment du type d'un objet, une expression conditionnelle choisit un comportement différent. Ce procédé rend la méthode originale abstraite puis déplace chaque branche de l'expression conditionnelle dans une méthode surchargée des sous-classes.

- Organisation des données

  Self encapsulate field : Crée des méthodes d'accès et de modifications des attributs et fait en sorte que l'accès à ces champs ne se fasse qu'en utilisant ces méthodes.

  Replace data value with object : Une variable est transformée en objet afin de contenir plus de données et gérer son propre comportement.

- Simplification des appels de methodes

Rename method : Renomme une méthode afin de mieux refléter le but de celle-ci.

Add parameter : Modifier une méthode nécessite parfois d'ajouter un paramètre parmi ceux déjà existant.

- Gestion de la généralisation

  Form template method : Si, dans une sous-classe, deux méthodes réalisent des opérations similaires dans le même ordre, mais que ces opérations sont différentes, ce procédé extrait les opérations dans des méthodes ayant les mêmes signatures afin que les méthodes originales deviennent les mêmes, puis utilise *pull up method* pour les remonter dans la superclasse.

  Pull up method : Remonte dans la superclasse, des méthodes ayant les mêmes résultats sur les sous-classes.

Exemples de refactorisation

La figure 4 illustre l'utilisation de la méthode *replace conditional with polymorphism* à partir de code extrait de la version originale du simulateur d'AVQ.
A la fin de la simulation, la liste des capteurs est parcourue pour en connaître l'état. Celui-ci est récupéré par la méthode *getEtat(Temps temps)* qui existent dans les deux classes de capteurs *CapteurAstable* et *CapteurMonostable*. Cette méthode renvoie l'état du capteur au *temps* donné en paramètre.

On constate que la même méthode (du moins le même nom de méthode et la même variable donnée en paramètre) est appelée pour les deux types de capteur mais qu'une conversion de type est employée après avoir différencié le type de capteur.

Bien que le traitement diffère suivant le type de capteur, il est possible de simplifier ce code en utilisant le polymorphisme. Ceci supprime la condition en abstrayant le type de capteur. Pour cela, le code commun aux deux types de capteur est d'abord regroupé dans

```
for(int j=0; j<VariablesGlobales.capteurs.size(); j++) {
        String cl = VariablesGlobales.tab_capteurs
        .elementAt(j).getClass().getName().toString();
        int stat= -1;
                if (cl.length()==15){
                    CapteurAstable cap = (CapteurAstable)
                    VariablesGlobales.capteurs.elementAt(j));
                    stat = cap.getEtat(temp);
                } else {
                     CapteurMonostable cap = (CapteurMonostable)
                     VariablesGlobales.capteurs.elementAt(j));
                     stat = cap.getEtat(temp);
                }
        ...
}
```

Figure 4 – Exemple de refactorisation (situation initiale)

une superclasse *Capteur* qui contiendra une méthode abstraite *getEtat(Temps temps)*. Les classes *CapteurMonostable* et *CapteurAstable* héritent de *Capteur* et implémentent la classe *getEtat(Temps temps)*. On évite ainsi une redondance de code dans les classes de type de capteur.

La figure 5 montre le résultat après modification :

```
for(int j=0; j<VariablesGlobales.capteurs.size(); j++) {
        Capteur cap = (Capteur) VariablesGlobales.capteurs.elementAt(j);
        int stat = cap.getEtat(temp);
}
```

Figure 5 – Exemple de refactorisation (situation finale)

1.3.3 Validation

Cette partie traite de la validité du modèle et des techniques de simulation. De nombreuses techniques existent pour valider l'exactitude d'un modèle. Cependant aucune technique universelle ne pourra conclure de la réussite ou de l'échec d'un modèle. Il est recommandé d'utiliser une combinaison de plusieurs techniques de validation pour augmenter la validité du modèle [18]. Parmi celles proposées par Sargent, trois techniques, décrites ci-dessous, conviennent au simulateur d'AVQ pour leurs capacités spécifiques de validation.

- *Event validity*

  Les occurrences du modèle de simulation sont comparées à celles du monde réel pour voir si elles sont similaires. Dans le simulateur d'AVQ, pour satisfaire à la technique *event validity*, l'agenda des AVQ doit être comparable aux habitudes de vie. Cela requiert de générer des données sur une longue période. Par exemple, un individu a l'habitude de boire du café cinq fois par semaine et du thé deux fois par semaine. Pour valider ces habitudes, des données sont générées sur un an de simulation. Les données résultantes doivent refléter les habitudes de vie citées ci-dessus.

- *Historical data validation*

  Cette technique consiste à comparer les données collectées dans le monde réel et les données obtenues par le simulateur pour déterminer le réalisme du comportement du modèle. Par exemple, des sorties capteurs sont recueillies dans l'environnement pendant que l'occupant réalise des AVQ. Les mêmes AVQ sont ensuite formulées dans un scénario soumis en entrée du simulateur. Celui-ci génère alors des sorties capteurs qui sont comparées à celles obtenues dans le véritable appartement. Cela suppose d'établir préalablement des critères de comparaison.

- *Animation*

  Le comportement du modèle est représenté graphiquement pour observer sa façon

d'évoluer en temps réel. Par exemple, une carte affiche la salle ou se déroule l'AVQ, met en évidence les capteurs actifs et affiche les valeurs des capteurs analogiques. Ce type d'animation est pratique pour vérifier la cohérence des données simulées.

Chapitre 2

Objectifs et méthodologie

Dans la version originale du simulateur, la lecture et la compréhension du code est très difficile. Un très grand nombre de classes ne représentent pas systématiquement un objet précis. Aucune hiérarchie de classes n'est apparente, des classes d'interfaces graphiques sont présentes mais incomplètes, certaines méthodes sont trop longues et complexes. Dans le but de faire évoluer le simulateur, il est nécessaire dans un premier temps de retravailler le code afin de faciliter ensuite l'étape de validation.

Ce chapitre établit les objectifs à atteindre ainsi que les moyens employés pour y parvenir. Ces objectifs concernent d'une part le code du simulateur, c'est-à-dire sa structure et sa lisibilité, et la validation des sorties générées par le simulateur d'autre part.

2.1 Objectifs

2.1.1 Objectif A – Maintenance et lisibilité du code

Étant donné la difficulté de modifier le code pour d'éventuelles améliorations futures, le premier objectif porte sur l'obtention d'une version du code plus lisible, plus facile à maintenir et qui reflète les aspects du monde réel. Nous l'avons divisé comme suit.

A.1 Améliorer la lecture du code et en faciliter la maintenance

- Améliorer la compréhension du code

  Rendre le code du simulateur plus lisible et accessible.

- Respecter les normes de programmation Java

  L'utilisation de normes de programmation permet aux programmeurs une meilleure assimilation du code et rend la maintenance du code par un tiers, plus aisée.

A.2 Retrouver l'organisation du monde réel dans les classes

- Hiérarchisation des classes selon les concepts qu'elles représentent

  Les classes doivent représenter les concepts suivants : la gestion des AVQ et du cycle de vie, la gestion des ressources, des capteurs, de l'environnement et du temps.

- Déterminer la spécificité de chaque classe

  Chaque classe doit correspondre à un objet précis et avoir un but.

2.1.2 Objectif B – Validation du simulateur

Le second objectif porte sur la validation du simulateur pour s'assurer que les sorties générées par le simulateur correspondent à la réalité. Nous désirons aborder la validation sous deux aspects distincts.

B.1 Visualiser graphiquement les sorties du simulateur

L'évolution des principaux concepts du simulateur (le cycle de vie des activités, l'utilisation des ressources, l'activation des capteurs, l'utilisation de l'environnement par l'individu simulé) doit pouvoir être observée graphiquement.

B.2 Comparer les sorties du simulateur avec une scène issue du monde réel

Les sorties capteurs générées doivent être comparées avec celles obtenues en milieu réel.

2.2 Méthodologie

Dans la partie suivante, nous établirons les techniques et la marche à suivre pour atteindre les objectifs cités précédemment.

2.2.1 Modification du code

Pour atteindre les objectifs de modification de code, les méthodes suivantes sont appliquées.

Isoler le code du simulateur : Dans un premier temps, il est nécessaire de rendre le code plus clair. Le code du simulateur étant mélangé à des classes d'interfaces graphiques, celui-ci doit être isolé pour ne garder que les classes essentielles au fonctionnement du simulateur.

Respecter les normes Java : Pour faciliter la lecture du code par les prochains développeurs du simulateur, le respect de certaines normes est nécessaire. Ces normes sont d'une part, relatives au langage de programmation, en l'occurrence Java. Elles concernent les noms de classes (première lettre en majuscule, les autres lettres en minuscule et le nom doit correspondre au but de la classe), le principe d'encapsulation doit être appliqué (l'accès aux attributs des classes devrait, dans la majorité des cas, être privé ou protégé, des méthodes de type *getter* et *setter* doivent permettre l'accès si nécessaire), des commentaires doivent être présents pour expliquer le but d'une classe, d'une méthode ou d'une variable.

D'autre part, des critères établis par le laboratoire permettent d'homogénéiser le code et d'en assurer la portabilité sur différents types de plateforme. Parmi ces critères, on retrouve la séparation du code, des fichiers de ressources (images, fichiers XML, etc.) et des fichiers de configuration, l'utilisation de chemins d'accès dans des fichiers de configuration, l'utilisation de fichiers XML ou de bases de données accessibles à partir de n'importe quelle plateforme pour les données.

Regrouper les concepts en package : Ranger les classes en package facilite la lecture du code et permet de regrouper les classes par concept. Chaque concept du simulateur (AVQ et cycle de vie, ressources, capteurs, etc.) doit être représenté par un ensemble de classes contenues dans un package explicite.

Appliquer les principes de refactorisation : Les principes de refactorisation vus au premier chapitre, offrent un cadre pour s'assurer une réécriture formelle. L'objectif A peut être atteint grâce à certains mécanismes de refactorisation ; les mécanismes comme *extract method*, *substitute algorithm* et *replace method with method object* permettent de réduire et de simplifier les méthodes trop longues, les mécanismes comme *move method*, *move field* et *extract class* vont permettre de déplacer du code pour redonner aux classes leur spécificité. Ces déplacements permettront également de réorganiser les classes pour refléter les concepts du monde réel. Ces objectifs peuvent également être atteint par d'autres mécanismes de refactorisation.

Appliquer les design patterns : L'emploi de *design patterns* s'associe parfaitement au processus de refactorisation. La simplification du code passe par certains principes de programmation que l'on retrouve dans les *design patterns*. La représentation d'ensemble comme la composition des zones de l'environnement peut s'implémenter avec le modèle *composite*. Le cycle de vie des activités peut être représenté par un *state design pattern*.

Ajout de nouvelles fonctionnalités : Les modifications du code, dans le but d'en simplifier sa lisibilité, ne modifie en rien le comportement du simulateur ou les sorties générées. Certaines fonctionnalités, telle que la composition hiérarchique d'activités ou la gestion des ressources lors d'une interruption, doivent être implémentées.

2.2.2 Techniques de validation

Pour atteindre les objectifs de validation, les techniques de validation suivantes sont appliquées.

Validation graphique : Pour valider graphiquement les sorties du simulateur, nous souhaitons disposer d'une interface permettant de suivre l'évolution de la simulation au cours du temps. Les activités et leurs états selon le cycle de vie, les ressources, les paramètres et les déplacements de l'individu simulé doivent pouvoir être visualisés. Un outil de gestion du temps doit être implémenté pour contrôler l'avance rapide, le retour en arrière ou mettre en pause le visionnement de la simulation rendant possible le contrôle de l'ensemble du visionnement de la simulation mais aussi le contrôle du temps d'une vue par rapport aux autres. Cet outil favorisera la comparaison entre des activités simulées à des temps différents, voire des activités provenant de scénarios différents. L'interface doit donc prendre en charge le visionnement de scénarios différents en même temps.

La conception de l'interface passe par l'élaboration d'un prototype *lo-fi* [16]. Cette version simpliste et en papier de l'interface a pour avantage de présenter les vues, l'organisation et les enchaînements possibles de l'interface et de les corriger si nécessaires, au fur et à mesure des itérations. Des tests utilisateurs doivent être effectués après chaque itération pour valider la cohérence et l'utilisation de l'interface. Ces utilisateurs peuvent être choisis parmi des étudiants du laboratoire, familiers au domaine de recherche mais n'ayant pas travaillé sur le simulateur d'AVQ.

Comparaison avec le monde réel : La validation du simulateur doit également passer par une comparaison avec le monde réel. Pour ce type de validation, nous appliquerons la technique *historical data validation* sur les sorties capteurs issues du monde réel et générées par le simulateur.

Lors de la réalisation d'une activité dans le laboratoire, les sorties capteurs peuvent

être enregistrées dans un fichier sous forme d'événements (à un temps précis, le serveur a reçu un signal d'activation de tel capteur). Cette même activité décrite dans un scénario et donnée comme entrée du simulateur produit en sortie l'agenda des activités, des ressources et des paramètres. Les paramètres qui sont associés à des capteurs permettent ensuite de générer des sorties capteurs sous la forme d'événements.

Pour effectuer cette comparaison, nous partons d'une activité courte (deux à trois minutes) que nous réalisons dans le laboratoire. Cette activité doit impliquer l'usage de ressources, l'activation de capteurs (donc le changement d'état de paramètres liés aux ressources et aux capteurs) et doit être organisée en sous-activités. Une fois l'activité définie, on demande à un volontaire de la réaliser dans le laboratoire tout en enregistrant les sorties capteurs et en filmant l'activité. Le film permet de reproduire le scénario correspondant à l'activité interprétée, pour être donné en entrée du simulateur et générer les sorties capteurs. Nous disposons ainsi de sorties capteurs issues du monde réel et du simulateur correspondant à une même activité. Nous pouvons ensuite comparer les deux séries de sorties capteurs grâce à l'interface de validation en observant les activations successives des capteurs. Ces observations sont faites selon plusieurs critères :

- correspondance entre l'activation d'un capteur avec l'activité en cours (et les changements d'état des paramètres),
- correspondance entre l'activation d'un capteur issu du monde réel et généré par le simulateur (laps de temps de différence entre les deux activations),
- logique de l'enchaînement des activations des capteurs.

Chapitre 3

Résultats

A partir de la méthodologie abordée précédemment, les résultats obtenus en matière de modifications du code d'une part, et de validation du modèle de simulation d'autre part, seront présentés dans ce chapitre.

Le code du simulateur a été amplement transformé grâce à la refactorisation du code, l'application de normes de programmation Java, l'utilisation de *design patterns* et l'ajout de nouvelles fonctionnalités. Les résultats de cette transformation partent de la compréhension des classes initiales pour finir avec les diagrammes des classes modifiées.

La validation du modèle de simulation nécessite une composition de plusieurs techniques. La première abordée est celle par animation graphique, concrétisée par le développement d'interfaces permettant de suivre l'évolution de la simulation. Puis une expérience réalisée dans l'environnement réel sera comparée avec les sorties d'un scénario simulé correspondant.

3.1 Résultats de refactorisation

3.1.1 Code original épuré

Plus d'une quarantaine de classes composent le code original du simulateur. On y trouve celles essentielles au simulateur ainsi que les interfaces de saisie de données. Parmi les entrées du simulateur, une petite partie, relative à l'environnement, est stockée dans une base de données *Microsoft Access* ©, les autres étant directement écrites dans le code.

Les interfaces, n'étant pas entièrement fonctionnelles car non terminées et non reliées à la base de données, sont d'abord mises de côté pour isoler le code du moteur du simulateur. Les données stockées dans la base de données sont recopiées dans le code de sorte à avoir toutes les données d'entrée regroupées au même endroit. Les données pourront par la suite être stockées dans des fichiers XML ou dans une une base de données. Cependant, les objets composant le simulateur seront modifiés lors de la refactorisation et il sera plus facile de réorganiser les données d'entrée une fois la refactorisation terminée.

Après avoir isolé le code du simulateur, les classes restantes de la programmation originale du simulateur sont les suivantes

- *Capteurs, CapteursAstables, CapteursMonostables*

  Ces classes contiennent les attributs et les méthodes relatifs aux capteurs. CapteursAstable et CapteursMonostables héritent de Capteurs et contiennent toutes deux une méthode setEtat(...). Cependant, cette méthode n'est pas abstraite dans la classe mère et nécessite l'emploi d'un *cast* lors du parcours de la liste des capteurs.
- *Compare, Outils*

  La classe Compare sert à la comparaison des ressources d'une tâche courante avec celles nécessaires à une tâche interruptrice. Outils contient des méthodes telles qu'une méthode de conversion de chaînes de caractères vers des variables numériques, des méthodes de connexion à une base de données et des méthodes

d'indentation de texte pour l'affichage.

- *ConditionsElementaires*

  Cette classe s'assure du respect des conditions pour qu'un paramètre système passe d'un état à un autre.

- *Globale_Horloge, Temps*

  Temps est un objet de gestion du temps (heures, minutes et secondes) tandis que Globale_Horloge gère le temps de la simulation en cours.

- *RessourcesSysteme, RessourcesSystemeRequises*

  Ces classes représentent les ressources disponibles dans l'environnement et celles qui sont requises pour la réalisation des tâches.

- *Taches*

  Cette classe représente les tâches accomplies ou à accomplir et contient les caractéristiques comme le temps d'exécution et la durée de validité.

- *ParametreSysteme*

  Les paramètres système décrivent les statuts des ressources dont l'état peut être détecté par des capteurs, c'est-à-dire le robinet est *ouvert* ou *fermé*.

- *Transitions, Evenements*

  Ces classes contiennent respectivement les changements d'états des tâches et des paramètres système, et sert d'historique de la simulation.

- *VariablesGlobales*

  Cette classe contient des variables utiles à la simulation et accessibles publiquement sans méthode d'accès de type *getters* et de *setters*. Parmi ces variables, on retrouve la liste des tâches, la liste des ressources et la description de l'environnement.

- *Simulateur*

  Simulateur est la classe principale. Elle contient la méthode *main*, la méthode *traitement* qui détermine les état suivants de chaque tâche et diverses méthodes d'affichage.

3.1.2 Constats

Après une première analyse du code original, certains constats ont été dressés, parmi lesquels l'accessibilité et la complexité de la méthode principale. Cette partie justifie ces constats et décrit la solution pour y remédier.

Accessibilité

Un des premiers constats sur le code original est l'utilisation de variables et d'attributs de classe de manière *friendly* sans utilisation de méthodes d'accès de type *getters* et *setters*. Le danger d'une telle utilisation réside dans la possibilité d'affecter une valeur sans en vérifier la validité. Selon le principe d'encapsulation, une classe ne devrait rendre publique que ce qui est nécessaire pour son utilisation.

Il est donc nécessaire d'appliquer ce principe pour redonner un accès logique aux attributs et aux méthodes de chaque classe. Les attributs qui nécessitent d'être accessibles depuis l'extérieur de la classe qui les contient devront l'être par des méthodes d'accès de type *getters* et *setters* pour en assurer la validité.

La classe nommée *VariablesGlobales*, qui contient une multitude d'attributs en libre accès, doit être remplacée par une classe contenant les paramètres de la simulation tels que la liste des tâches à simuler, la liste des ressources et la description de l'environnement. La classe résultante est la classe *SimulationParameters* présentée dans la section 3.1.3.

Méthode principale

La classe simulateur contient une méthode *traitement* qui gère les tâches au cours de la simulation. Tant que la simulation n'est pas terminée, la liste des tâches est parcourue plusieurs fois pour déterminer le prochain état de chaque tâche selon le cycle de vie des activités.

Le code de la figure 6 est l'une des boucles de la méthode *traitement*. Dans ces boucles,

```
for(int i=0 ; i<lestaches.size() ; i++) {
        Taches ta = (Taches) lestaches.elementAt(i) ;
        if (((ta.getEtat()=="En attente") || (ta.getEtat()=="Non prévue"))
        &&(ta.conditionTemporelle()) && (ta.conditionInitiale())
        &&(conditionConflitPosition(ta)) && (ExisteAssezRessource(ta))) {
            tache.setEtat("En exécution") ;
            ...
        }
}
```

Figure 6 – Exemple de parcours de la liste des tâches

ce sont toujours les tâches qui sont parcourues, seules les conditions pour déterminer le prochain état de la tâche varient. Ces conditions dépendent principalement de l'état actuel de la tâche. Ainsi, pour décider de mettre une tâche en exécution, la boucle du code de la figure 6 commence par vérifier s'il s'agit d'une tâche interruptrice ou en attente, avant de vérifier les conditions temporelles, les ressources et d'autres conditions. Cette méthode est donc très longue et très complexe.

Ces vérifications liées à l'état actuel de la tâche pourraient être effectuées dans une classe représentant l'état actuel de la tâche, c'est-à-dire que chaque tâche doit contenir un attribut *état* relatif à son état courant. Pour cela, l'utilisation d'un *state design pattern* est recommandée.

Le *state design pattern* place le comportement associé à un état particulier dans un objet [9]. Appliquer ce patron de conception revient donc à créer un objet état pour chaque état du cycle de vie d'une activité comme le montre le diagramme des classes des activités et du cycle de vie dans la section 3.1.3.

3.1.3 Diagramme des classes

A la suite du processus de refactorisation, le code est organisé en plusieurs paquets représentant les aspects importants du simulateur tels que les activités et le cycle de

vie des activités, l'environnement, les ressources, les capteurs, les classes principales, les classes de chargement et de sauvegarde des simulations et des classes permettant le suivi de la simulation. La répartition des classes qui en résulte est expliquée dans les parties suivantes.

Activités et cycle de vie

Les activités dans le simulateur original étaient représentées par un vecteur de tâches et la notion d'AVQ n'était pas encore implémentée. Afin de prendre en compte le principe d'AVQ (une activité peut être composée de sous activités et d'actions), la classe originale *Tache* est devenue la classe abstraite *Activity* dont héritent les classes *Task* et *Action*. Ceci permet la création d'une hiérarchie d'activités. Ainsi, l'activité *Faire le café* peut être composée de sous-activités telles que *Prendre une tasse* et *Faire bouillir de l'eau*, elles-mêmes composées de sous-activités et d'actions.

Il est également important de situer une activité par rapport à ses activités supérieures et précédentes. Les activités supérieures sont celles qui se situent hiérarchiquement au-dessus et qui doivent être en cours d'exécution pour que l'activité puisse être exécutée, tandis que les activités précédentes sont les activités dont l'exécution doit être achevée pour que l'activité courante puisse commencer son exécution. Les attributs *upperActivity*, *subActivities* permettent à l'activité de se situer par rapport à la hiérarchie de l'AVQ et *priorActivities* sert à lister les activités précédentes.

Un des autres changements entre le code original et le code actuel concerne la gestion du cycle de vie des activités. Le comportement d'une activité dépend essentiellement de son état parmi les neuf états possibles présentés dans la section 1.2.3. Pour alléger les tests conditionnels qui vérifiaient l'état de l'activité en cours ainsi que les états des autres activités pour déterminer le prochain état de chaque activité, le *state design pattern* a été utilisé. Ce modèle de conception de type comportemental permet d'assigner ou de déléguer des responsabilités aux objets. Ces responsabilités sont le cheminement des

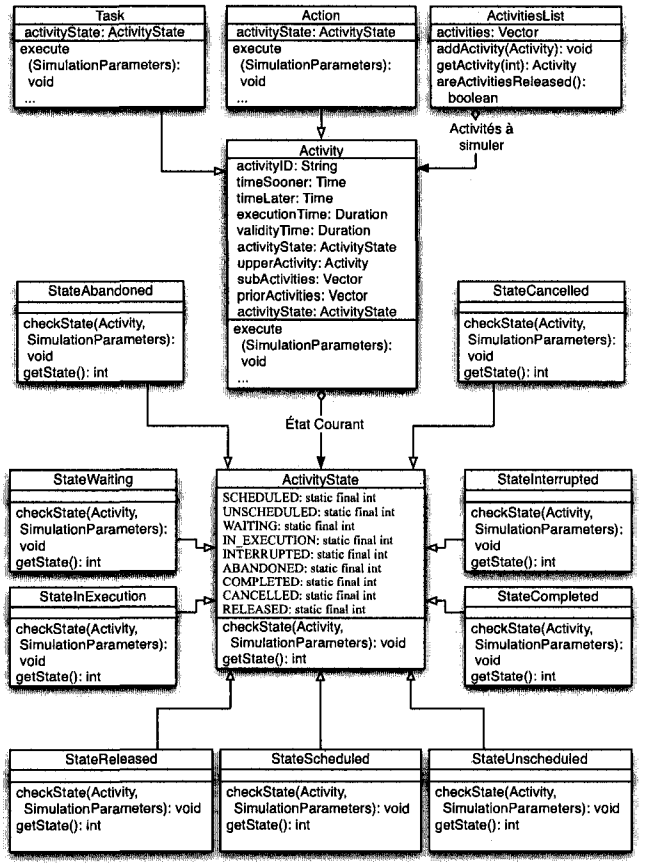

Figure 7 – Gestion des activités et du cycle de vie

activités et le respect du cycle de vie. Chaque activité contient un attribut *ActivityState* dont la classe abstraite contient une méthode qui vérifie l'état de l'activité à chaque tour d'horloge et le modifie si nécessaire. Chacun des neuf états du cycle de vie est représenté par un objet héritant de la classe *ActivityState* et implémentant la méthode de vérification de l'état. Les tests conditionnels de la méthode *traitement* ont été considérablement simplifiés et répartis à travers les neuf états concrets du modèle de conception.

Chaque activité contient une méthode *execute* appelée à chaque tour d'horloge. La

consommation des ressources est gérée lors de l'appel à cette méthode. Les activités et les interruptions à simuler sont stockées dans deux instances distinctes de la classe *ActivitiesList*. La figure 7 représente le diagramme des classes relatives à la gestion hiérarchique des activités et de leur cycle de vie.

Gestion de l'environnement

L'environnement pour le simulateur d'activités consiste en une description des zones de l'appartement où nous souhaitons réaliser les simulations. Cette description contient la configuration des zones ainsi que les transitions et le temps de transition entre ces zones.

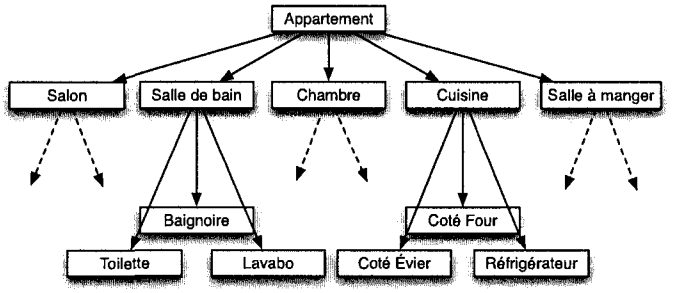

Figure 8 – Description de l'environnement

Dans le code original du simulateur, l'environnement de simulation est représenté par une liste de zones et une matrice de transitions entre ces zones. Il est possible de diviser l'appartement en pièces comme le salon ou la cuisine. Cependant, si une pièce devait être à son tour divisé en zones, cette division faisait disparaître l'appartenance de ces zones à la pièce, par exemple l'environnement comprenait soit la cuisine comme une zone, soit des zones dont l'ensemble composait la cuisine. Cependant, dans le dernier cas, il n'était plus possible de reconnaître la cuisine comme une pièce.

Pour y remédier, un modèle de conception de type structurel a été utilisé. Le modèle

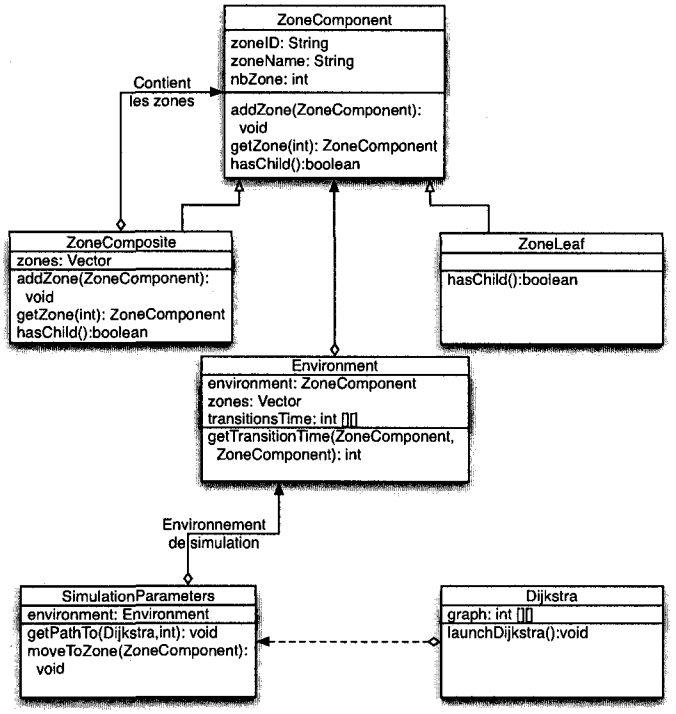

Figure 9 – Gestion de l'environnement

composite permet de modéliser les zones dans une structure arborescente rendant possible une description de l'environnement sur plusieurs niveaux comme illustré dans la figure 9. L'appartement peut donc être divisé indéfiniment en zones. La figure 8 décrit une configuration possible de l'environnement. L'appartement est divisé en cinq zones représentant les cinq pièces principales de l'appartement, dont deux pièces divisées en trois zones.

Selon l'activité à réaliser, on peut choisir le niveau de description de l'environnement, ainsi lorsque le patient fait cuire un plat, on peut choisir la zone *coté four* ou lorsqu'il lave le sol de la cuisine, on choisira la zone *Cuisine* lors de la simulation.

Gestion des ressources

Le simulateur dispose de la liste des ressources disponibles pour les besoins de la simulation. Cette liste contient les ressources liées à l'environnement et celles liées à la personne. Chaque ressource est représentée par son identifiant, son nom et sa quantité totale dans la classe *Resource*.

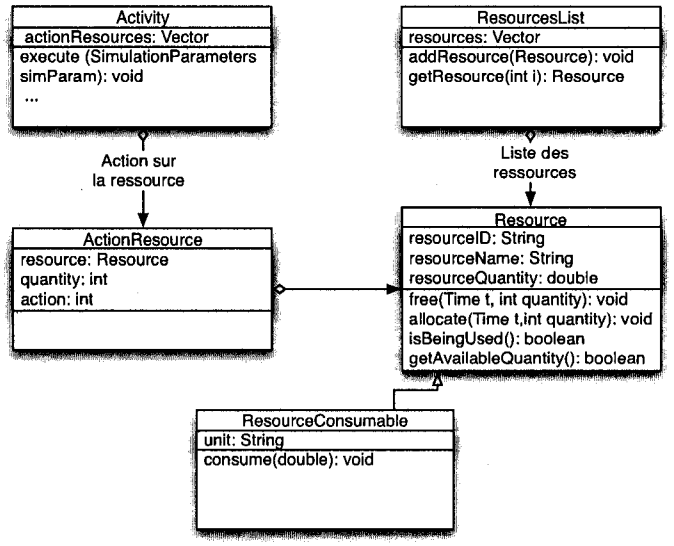

Figure 10 – Gestion des ressources

Dans le cas des ressources consommables, l'unité de mesure et l'utilisation sont également pris en compte grâce à la classe *ResourceConsumable*. Une ressource peut être utilisée totalement ou en partie. Dans le dernier cas, le reste de la ressource est disponible pour d'autres activités.

Lorsque que l'activité entre en exécution, elle provoque une interaction avec les ressources dont elle dépend. Cette interaction est décrite grâce à la classe *ActionResource* qui lie l'activité à compléter, avec une ressource et sa quantité requise. Les trois actions possibles sont l'*allocation* et la *libération* pour tous les types de ressources, et l'*utilisation*

dans le cas des ressources consommables. L'*allocation* et la *libération* sont réalisées respectivement au début et à la fin de l'activité tandis que la *consommation* est simulée à chaque tour d'horloge lors de l'appel de la méthode *execute* de la classe *Activity*. La consommation est possible uniquement pour les ressources consommables représentées par la classe *ResourceConsumable* qui hérite de *Resource*. La figure 10 présente les classes nécessaires à la gestion des ressources et leur cohésion avec la classe *Activity*.

Gestion des paramètres et des capteurs

Lors de la simulation d'une activité, celle-ci peut modifier l'état d'un objet ou d'une ressource pour mener à bien son exécution. La modification de ces états est réalisée grâce aux paramètres.

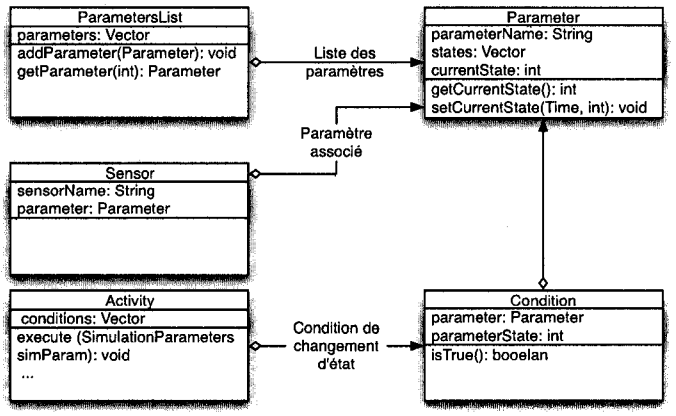

Figure 11 – Gestion des paramètres et des capteurs

Le simulateur dispose de la liste des paramètres dont on désire connaître l'évolution. Ces paramètres sont définis par un nom, par une liste d'états possibles et par l'état courant du paramètre. Le premier paramètre de la liste correspond à la position du patient dont les états possibles sont les zones décrites dans l'environnement. La figure 11 illustre les classes responsables de la gestion des paramètres et des capteurs.

Si l'exécution d'une activité nécessite qu'un paramètre ait un état précis, la liste des conditions contenues dans la classe *Activity* précise chaque état à atteindre pour tous les paramètres concernés. Si la condition est vraie, l'état souhaité du paramètre correspond à l'état courant de ce paramètre, il n'y a aucun changement. Autrement, l'état du paramètre est modifié grâce à la méthode *setCurrentState* qui prend également en paramètre le temps auquel la modification a eu lieu. S'il est nécessaire de connaître les sorties capteurs pour un paramètre ne comprenant que deux états, la classe *Sensor* retrouve, dans le paramètre qui lui est associé, chaque modification de l'état du paramètre au cours du temps.

Gestion de la simulation

La figure 12 illustre le fonctionnement du simulateur. Le scénario de simulation qui regroupe les entrées du simulateur, c'est-à-dire le temps de simulation, l'environnement, les ressources, les paramètres, les activités et les interruptions, est chargé dans la classe *SimulationParameters*. Cette dernière sert de liaison entre le simulateur et les objets nécessitant l'accès aux différents composants du scénario.

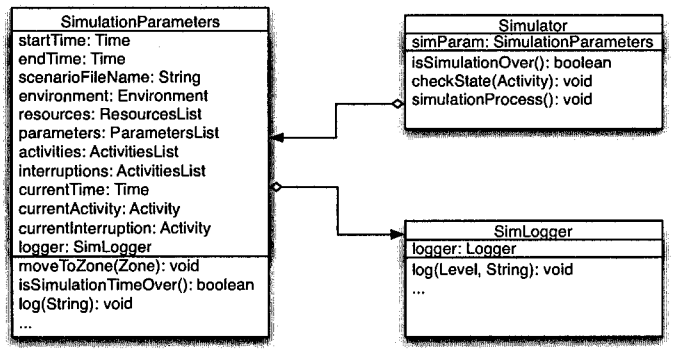

Figure 12 – Gestion de la simulation

Une fois tous les paramètres de simulation chargés, la classe *Simulator* peut lancer le

processus de simulation grâce à la méthode *simulationProcess*. Celle-ci appelle la méthode de vérification d'état *checkState* pour chaque activité et chaque interruption à chaque tour d'horloge jusqu'à la fin de la simulation. La simulation est terminée lorsque toutes les activités sont finies ou lorsque le temps de simulation est écoulé.

La classe *SimLogger*, dont une instance est accessible depuis les paramètres de simulation, permet de garder une trace des changements lors du déroulement de la simulation comme l'utilisation des ressources, les changements d'état des paramètres, des activités et des interruptions. Ces traces sont affichées en temps réel à l'écran ou dans un fichier.

Gestion des transitions

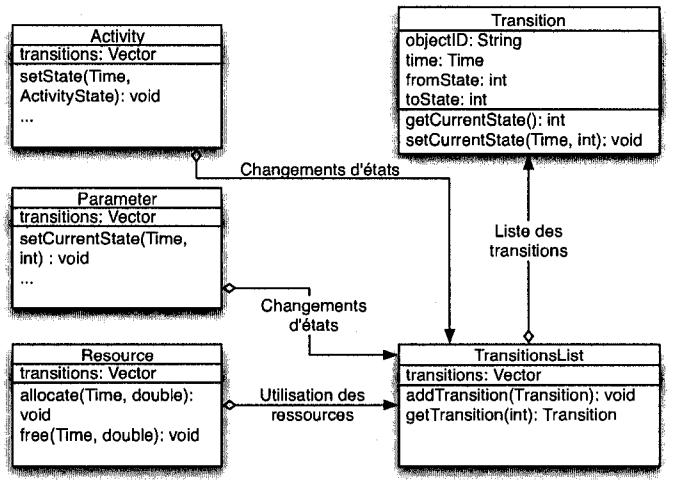

Figure 13 – Gestion des transitions

Dans la description des réseaux de Petri, lorsque les préconditions d'une activité sont réunies telles que le temps de départ de l'activité et les ressources disponibles, une transition est déclenchée. Les postconditions représentent les résultats de l'activité, c'est-à-dire activité réalisée, ressource libérée ou consommée. De ce fait, cette transition marque

la réalisation d'un changement. Les classes *Transition* et *TransitionsList* contiennent et listent les transitions pour les activités, les paramètres et les ressources.

A la fin de la simulation, l'ensemble de ces transitions représentent l'évolution de la simulation. Ce sont ces données qui sont sauvegardées comme l'un des agendas issu du scénario simulé. Les transitions des paramètres sont également utilisées pour reproduire les sorties des capteurs associés à ces paramètres.

3.1.4 Apport de la refactorisation sur le code

Le processus de refactorisation a permis de modifier profondément la structure du code original pour obtenir le diagramme des classes présenté précédemment. En comparaison avec le code original, les normes Java sont mieux respectées, la lecture du code est facilitée par la correspondance entre le nom des classes et leur but, et par le regroupement des classes en *packages*. Cette réorganisation des classes constitue également une meilleure représentation du monde réel. Par exemple, l'organisation hiérarchique des AVQ en tâches et actions est dorénavant respectée. Le cycle de vie des activités, qui n'était pas du tout visible dans la structure initiale du code, apparaît maintenant clairement dans le *package* des activités. Cette amélioration est due à l'application du *state design pattern* qui, par une délégation du comportement des activités à chacun des neufs états, permet de visualiser, dans le diagramme des classes, l'ensemble des états du cycle de vie des activités. De même l'environnement était représenté par un vecteur de zones. Un *composite design pattern* permet à présent de découper indéfiniment les zones en sous-zones pour augmenter le niveau de détail de la description de l'environnement.

3.1.5 Nouvelles fonctionnalités implémentées

Parmi les nouvelles fonctionnalités implémentées lors de la refactorisation du code, on retrouve les suivantes.

- Décomposition hiérarchique des AVQ

  Alors que le simulateur initial ne gérait qu'un vecteur de tâches, il est désormais possible de décrire les activités à simuler sous la forme d'AVQ en la décomposant en tâches, sous-tâches et actions.

- Description de l'environnement

  L'environnement peut être décrit en un nombre indéfini de zones, qui peuvent être composées en un nombre indéfini de sous-zones, etc. Cette nouvelle description laisse, à l'utilisateur du simulateur, la possibilité de découper facilement une pièce de l'appartement en zones. Par exemple, dans la cuisine, on retrouve le coté du réfrigérateur, la zone évier, le comptoir, etc.

- Consommation des ressources

  Les ressources de types consommables ont été ajoutées dans le *package* ressource pour permettre de suivre l'évolution de la consommation d'une ressource pendant la simulation. L'exécution d'une activité qui consomme ce type de ressource, engendre la diminution de la quantité de cette ressource.

- Gestion des fichiers XML

  Les données, auparavant stockées directement dans le code ou dans une base de données, sont réparties dans plusieurs fichiers XML. Ces fichiers de configuration du simulateur contiennent les AVQ à simuler, les ressources mises à disposition, les paramètres à observer et la description de l'environnement. Ces fichiers sont chargés au début de la simulation. Ces fichiers sont directement modifiables par l'utilisateur du simulateur qui n'a plus besoin de toucher au code pour modifier les entrées du simulateur. En fin de simulation, les transitions observées sont écrites dans un fichier XML.

3.2 Résultats de validation

3.2.1 Data historical validation

Cette partie présente le mode opératoire de l'expérience réalisée en laboratoire dans le cadre de la technique *data historical validation.*

Pour cette technique de validation, un scénario d'AVQ est choisi. Ce scénario comprend la liste des activités, des tâches et des actions que l'on souhaite observer et contient également la liste des ressources mises à disposition de l'occupant. Un volontaire réalise ces activités dans le laboratoire. Les sorties capteurs engendrées par ses actions sont enregistrées et la scène est filmée pour s'assurer de la concordance entre les sorties capteurs et les actions correspondantes.

Le même scénario est alors donné en entrée du simulateur d'AVQ et les sorties capteurs générées sont comparées avec celles obtenues dans le monde réel.

Ce scénario a été réalisé et filmé une première fois sans interruption, puis avec interruption. L'interruption consiste en un appel téléphonique auquel le patient répond lors de la préparation du plat.

Scène observée et simulée

La scène de référence est la réalisation du plat "Tomate-Mozarella". L'occupant est assis à la table de la salle à manger. Lorsqu'il se lève, il se dirige vers la cuisine pour sortir une tomate et du fromage du réfrigérateur. Il prépare le plat puis se rassoit à table pour manger.

Décomposition de la scène : Liste des tâches et des actions à exécuter par le volontaire lors de l'expérience.

- Lire dans la salle à manger
- Aller dans la cuisine
- Prendre l'assiette contenant la tomate et le fromage dans le réfrigérateur

- Poser l'assiette sur le comptoir
- Laver la tomate
- Prendre le couteau dans le tiroir
- Prendre une nouvelle assiette dans le placard
- Couper la tomate et le fromage et mettre dans l'assiette
- Aller dans la salle à manger, poser l'assiette sur la table et manger

Tableau 1 – Ressources disponibles

Ressource	Type	Quantité
Tomate	Consommable	1
Fromage	Consommable	1
Couteau	Matérielle	1
Assiette	Matérielle	2

Ressources disponibles : Le tableau 1 présente les ressources rendues disponibles pendant l'expérience. L'utilisation de la chaise, de la table et de l'eau est ignorée.

Tableau 2 – Listes des capteurs utilisés

Identifiant	Type	Emplacement
IR001	Infrarouge	Cuisine comptoir
IR004	Infrarouge	Cuisine évier
IR005	Infrarouge	Salle à manger
TC010	Tapis tactile	Réfrigérateur
TC012	Tapis tactile	Cuisine évier
EM001	Électro-Magnétique	Porte réfrigérateur
EM002	Électro-Magnétique	Placard (assiette)
EM003	Électro-Magnétique	Tiroir (couteau)

Capteurs et zones observés : La liste et l'emplacement des capteurs sont présentés dans le tableau 2. Cette liste comprend essentiellement des capteurs infrarouges, électromagnétique et des tapis tactiles.

La figure 14 montre la disposition des capteurs dans le laboratoire. Les capteurs utilisés sont foncés. L'expérience se déroule dans la salle à manger, la cuisine et le salon en cas d'interruption (téléphone).

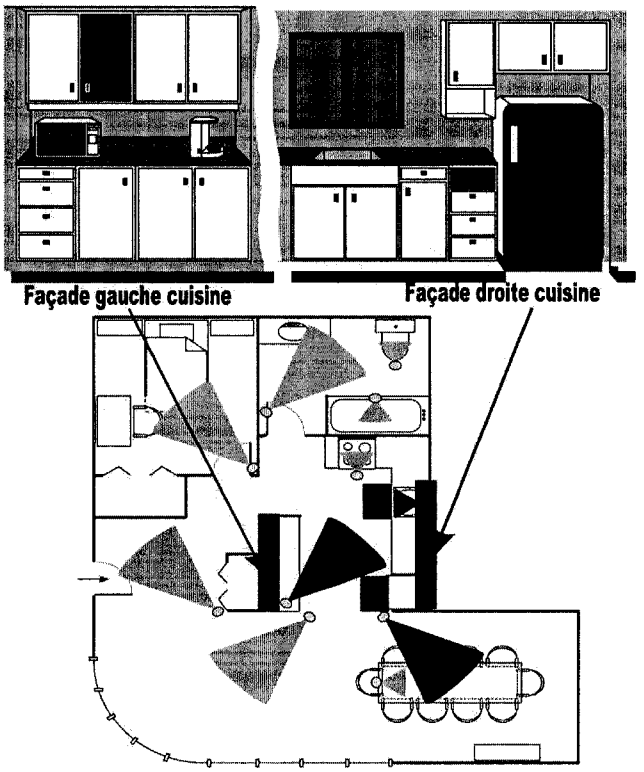

Figure 14 – Répartition des capteurs

Simulation de la scène observée : A partir de l'enregistrement vidéo de l'expérience en laboratoire, les événements observés sont retranscrits en AVQ, en tâches et en actions. L'environnement et les ressources sont décrits et les paramètres que l'on souhaite observer sont décrits dans les fichiers de configurations du simulateur.

Les entrées du simulateur sont réparties dans six fichiers de configuration.

– activities.xml

Contient les activités à simuler sous la forme d'AVQ, de tâches et d'actions, les

associations entre les activités et les ressources, et les associations entre les activités et les changements d'états des paramètres. La figure 15 contient un exemple de description de l'AVQ *Laver tomate* qui contient trois sous-activités dont une tâche et deux actions. Cette AVQ est prévue entre 14h37 et 15h, son temps de validité est d'une journée, son temps d'exécution est nul puisqu'il dépend du temps d'exécution de ses sous-activités. Les activités *A2_OuvrirRobinet* et *A2_FermerRobinet* sont associées à la ressource *R0_TAPK* qui correspond au robinet de la cuisine. Les actions engendrées par ces activités sont respectivement l'allocation puis la libération de la ressource. Ces mêmes activités sont également associées au paramètre *P1_TAPK* qui représente l'état du robinet, soit ouvert puis fermé.

- interruptions.xml

  Contient les interruptions à simuler sous la forme d'AVQ, de tâches et d'actions. La structure de ce fichier est identique à celle du fichier *activities.xml*.

- environment.xml

  Décrit les zones qui composent l'environnement, leurs éventuelles transitions et temps de transition. Une zone peut être une pièce de l'appartement, un ensemble de pièces ou une partie d'une pièce. Pour l'expérience, l'accent est mis sur la salle à manger, le salon et trois zones de la cuisine comme montré dans la figure 16.

- resources.xml

  Contient les ressources dont on souhaite observer l'évolution durant la simulation. La figure 17 montre les ressources et leurs quantités disponibles. Pour l'expérience, la tomate est considérée comme une ressource ordinaire, tandis que le fromage est considéré comme une ressource consommable. Il est donc possible d'observer sa consommation au cours du temps.

- parameters.xml

  Contient les paramètres dont on souhaite connaître l'évolution au cours du temps,

```
<activity activityid="A1_LaverTomate" type="task" timesooner="14 :37 :00"
timelater="15 :0 :0" executionTime="0 :0 :0" validityTime="24 :0 :0"
zoneid="ZL_SINK">Laver Tomate
    <activity activityid="A2_OuvrirRobinet" type="action">
    Ouvrir Robinet<activity>
    <activity activityid="A2_FermerRobinet" type="action">
    Fermer Robinet<activity>
    <activity activityid="A2_Egoutter" type="task"
    timesooner="14 :37 :00" timelater="15 :0 :0" executionTime="0 :0 :2"
    validityTime="24 :0 :0">Égoutter Tomate<activity>
<activity>

<resourceuse activityid="A2_OuvrirRobinet" resourceid="R0_TAPK"
quantity="1" action="1" >
<resourceuse activityid="A2_FermerRobinet" resourceid="R0_TAPK"
quantity="1" action="3" >

<conditionsensor activityid="A2_OuvrirRobinet" parameterid="P1_TAPK"
parameterstate="1">
<conditionsensor activityid="A2_FermerRobinet" parameterid="P1_TAPK"
parameterstate="0"/>
```

Figure 15 – Fichier de configuration : activities.xml

ainsi que la liste de leurs différents états possible. La figure 18 montre un exemple de paramètre qui comprend deux états, *ouvert* et *fermé*.

- scenario.xml

  Contient une description du scénario, le temps de départ et de fin de la simulation et les chemins des cinq fichiers XML précédent.

Comparaison entre les données simulées et la scène observée

A partir de la scène décrite précédemment, les données capteurs récoltées et la vidéo de la scène ont été comparées. Le tableau 3 illustre cette comparaison en montrant d'un coté, les observations faites à partir de la vidéo, et de l'autre, les transitions enregistrées

```
<zone type="zonecomposite" zoneid="ZC_LAB">Laboratoire Domus
    <zone type="zonecomposite" zoneid="ZC_KITC">Cuisine
        <zone type="zoneleaf" zoneid="ZL_CPT">Comptoir<zone>
        <zone type="zoneleaf" zoneid="ZL_SINK">Evier<zone>
        <zone type="zoneleaf" zoneid="ZL_FRID">Réfrigrateur<zone>
    <zone>
    <zone type="zoneleaf" zoneid="ZL_LIVR">Salon<zone>
    <zone type="zoneleaf" zoneid="ZL_DINR">Salle à manger<zone>
<zone>

<transitiontime zone1="ZL_FRID" zone2="ZL_CPT">3<transitiontime>
<transitiontime zone1="ZL_FRID" zone2="ZL_SINK">2<transitiontime>
<transitiontime zone1="ZL_CPT" zone2="ZL_SINK">2<transitiontime>
```

Figure 16 – Fichier de configuration : environment.xml

```
<resource resourceid="R0_TAPK" availableQuantity="1">
Robinet <resource>
<resource resourceid="R0_KNIF" availableQuantity="1">
Couteau <resource>
<resource resourceid="R1_TOMA" availableQuantity="1">
Tomate <resource>
<resource resourceid="R1_MOZA" availableQuantity="50"
unit="Grams"> Mozarella <resource>
```

Figure 17 – Fichier de configuration : resources.xml

en sortie du simulateur.

3.2.2 Validation par animation

Cette partie présente le mode opératoire du développement l'interface de validation. Les différentes vues développées y sont présentées et commentées.

```
<parameter parameterid="P1_TAPK">Robinet Cuisine
    <statuslist>
        <status>Fermé<status>
        <status>Ouvert<status>
    <statuslist>
<parameter>
```

Figure 18 – Fichier de configuration : parameters.xml

Objet de l'interface

L'interface de validation a pour but de faciliter la lecture des sorties du simulateur. Ces sorties représentent l'agenda de la simulation, c'est-à-dire l'historique de chaque changement d'état d'une activité, d'une ressource ou d'un paramètre.
L'utilisateur de l'interface doit être capable de charger un scénario, de sélectionner un ou plusieurs agendas correspondants, de suivre l'évolution des activités, des ressources et des paramètres dans le temps. Pour s'assurer que l'interface réponde à ces critères, un prototype *lo-fi* a été réalisé.

Prototypage

Un prototype *lo-fi* de l'interface de validation a été réalisé dans le cadre d'un projet de baccalauréat pour lequel Nadine Gagnon et Mickaël Beaudry ont activement participé. Le but de ce prototype était d'avoir un aperçu concret de l'interface de validation avant de la développer. Pour cela, les étudiants au baccalauréat ont dessiné sur papier plusieurs versions de l'interface afin de mettre au point une interface fonctionnelle en terme d'utilisabilité et répondant aux critères fixés. La dernière version papier est présentée à l'annexe A.

La création d'un prototype *lo-fi* inclut des tests-utilisateurs à la fin de chaque itération. Les premiers tests ont été effectués par nous-mêmes pour éliminer les principaux défauts d'utilisabilité. Les derniers tests ont été réalisés sur des personnes extérieures au

Tableau 3 – Comparaison entre les données simulées et la scène observée

Monde réel (vidéo)	Monde de la simulation (transitions)
14 :37 :06 Le sujet lit dans la salle à manger	14 :37 :07 Activité *Lire* : En attente -> En Exécution
14 :37 :20 Le sujet va dans la cuisine	14 :37 :20 Paramètre *Position* : Salle à manger -> Zone réfrigérateur
14 :37 :22 Ouverture du réfrigérateur	14 :37 :20 Paramètre *Porte Frigo* : Fermé -> Ouvert
14 :37 :31 Le sujet lave la tomate	14 :37 :32 Paramètre *Position* : Zone comptoir -> Zone évier 14 :37 :32 Paramètre *robinet cuisine* : Fermé -> Ouvert 14 :37 :32 Ressource *robinet cuisine* : Libre -> alloué
14 :37 :39 Le sujet prend l'assiette	14 :37 :40 Paramètre *Placard* : Fermé -> Ouvert
14 :37 :45 Le sujet prend le couteau	14 :37 :45 Paramètre *Tiroir* : Fermé -> Ouvert 14 :37 :46 Ressource *Couteau* : Libre -> alloué
14 :37 :49 Le sujet coupe la tomate	14 :37 :49 Activité *Couper la tomate* : En attente -> En Exécution
14 :38 :30 Le sujet retourne dans la salle à manger	14 :37 :30 Paramètre *Position* : Zone comptoir -> Salle à manger

projet du simulateur. Ceci permet de s'assurer d'une prise en main facile de l'interface par n'importe quel utilisateur. Des exemples de l'interface *lo-fi* se trouvent à l'annexe A.

L'interface, bien que complexe par le nombre de vues et les fonctionnalités à supporter au niveau de la gestion des scénarios et du temps, a été affinée grâce aux différentes itérations de prototypage. Les tests-utilisateurs ont permis de stabiliser la gestion du temps, notamment la synchronisation du temps entre les différentes vues. Guidée par les commentaires des testeurs, la synchronisation du temps a été corrigée et simplifiée pour permettre de lier des vues au même temps ou à des temps différents, tout en lançant le

déroulement du temps instantanément pour toutes les vues ouvertes.

De plus, les tests ont démontré l'intérêt d'utiliser des couleurs pour distinguer les scénarios auxquels les vues sont associées lors du visionnement de plusieurs scénarios. Finalement, ces tests ont démontré une bonne compréhension des différentes vues et de leurs rôles de la part des utilisateurs. Ces vues sont expliquées plus en détail dans la section suivante.

Vues de l'interface de validation

A partir du prototype *lo-fi*, l'interface de validation a été décomposée en plusieurs fenêtres.

- Gestionnaire de fenêtres

  Permet d'ouvrir un scénario de simulation et parcourir les différents agendas qui en découlent. Un ou plusieurs agendas peuvent être chargés en même temps. Pour chacun des agendas chargés, l'utilisateur peut ouvrir cinq vues différentes ; la vue des AVQ, des ressources, des ressources personnelles, des paramètres et le plan du laboratoire.

- Gestionnaire du temps

  Bien que chacune des vues suivantes comprennent une barre de glissement pour naviguer dans le temps, ces dernières sont indépendantes. Cette fenêtre permet de déclencher le déroulement du temps des vues ouvertes, qu'elles soient toutes au même temps ou non.

- Vue générale des AVQ

  Permet de visualiser l'agenda des AVQ. La liste des AVQ, des tâches et des actions est affichée avec leur état (selon le cycle de vie) et le temps de la dernière modification comme indiqué sur la figure 19. Des filtres permettent de sélectionner l'affichage des AVQ, des tâches, des actions et des interruptions. Il est également possible de sélectionner certaines activités pour les afficher dans la vue détaillée des

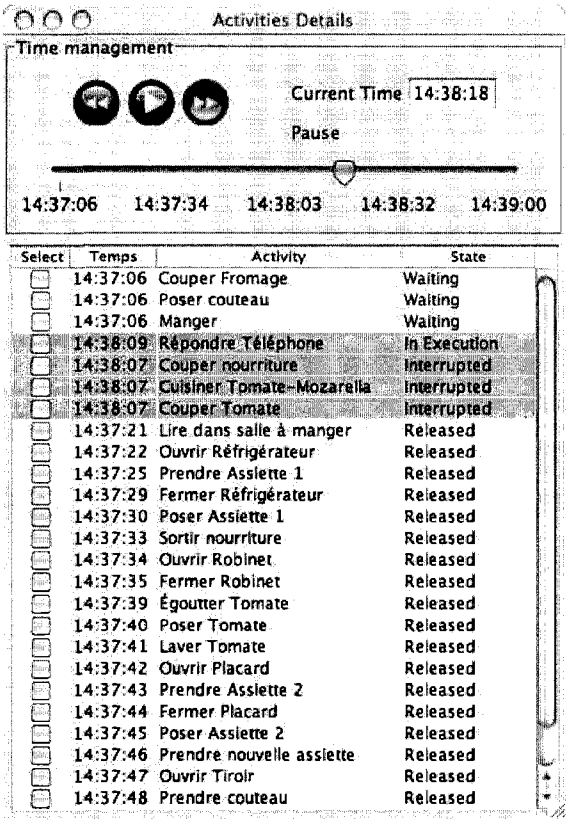

Figure 19 – Vue générale des AVQ

AVQ.

- Vue détaillée des AVQ

  Permet de visualiser les activités sélectionnées dans la vue des AVQ de façon déatillée. Les activités sont représentées hiérarchiquement sous forme d'arbre. La couleur de la branche indique l'état actuel de l'activité (selon le cycle de vie).

- Vue générale des ressources physiques et personnelles

  Permet de visualiser l'utilisation des ressources au cours du temps. Les ressources sont affichées avec leur état (allocation, libération ou consommation), leur quantité disponible et le temps de la dernière modification. La vue des ressources physiques

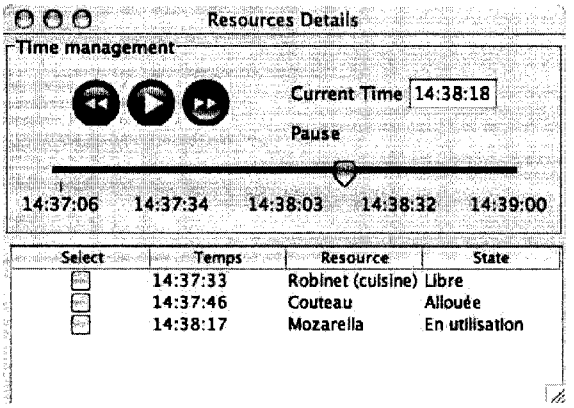

Figure 20 – Vue générale des ressources

et personnelles sont identiques mais sont affichées dans deux fenêtres distinctes. La figure 20 montre la vue générale des ressources physiques.

- Vue détaillée des ressources physiques et personnelles

  Permet un affichage détaillé de l'utilisation des ressources dans le temps pour les ressources sélectionnées dans la fenêtre précédente. Chaque changement survenant pendant la période de simulation est affiché sur un même graphique. Lorsque la ressource est consommable, un second graphique affiche la quantité restante. La vue détaillée des ressources physiques et personnelles sont identiques mais sont affichées dans deux fenêtres distinctes. La figure 21 montre la vue détaillée des ressources physiques.

- Vue générale des paramètres

  Cette vue montre la liste des paramètres à observer, leur dernier état et le temps auquel ce dernier état a été observé. Les paramètres représentent les événements qui peuvent être engendrés par des capteurs dans le monde réel comme montré sur la figure 22. Tout comme les vues des ressources et des AVQ, certains paramètres peuvent être sélectionnés pour être affichés dans la vue détaillée.

- Vue détaillée des paramètres

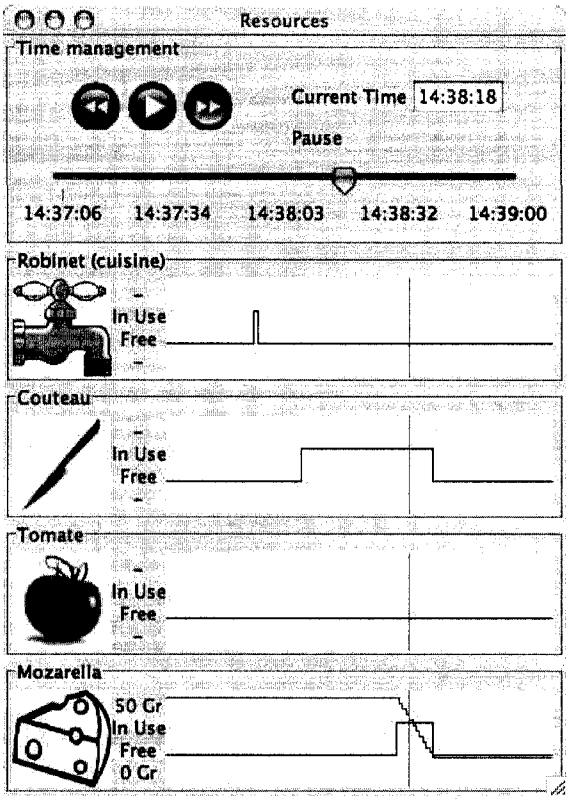

Figure 21 – Vue détaillée des ressources

Permet un affichage détaillé des changements d'état des paramètres. Chaque changement survenant pendant la période de simulation est affiché sur un même graphique comme indiqué sur la figure 23.

- Plan de l'appartement

  Le plan du dessus de l'appartement est affiché dans cette fenêtre pour faciliter le suivi du paramètre *position du patient.* Les états de ce paramètre peuvent être mise en correspondance avec des capteurs de localisation du patient, tels que les tapis tactiles et les capteurs infrarouges, comme indiqué sur la figure 24. Cette vue permet de suivre l'activation de ces capteurs en fonction du paramètre *position du patient.*

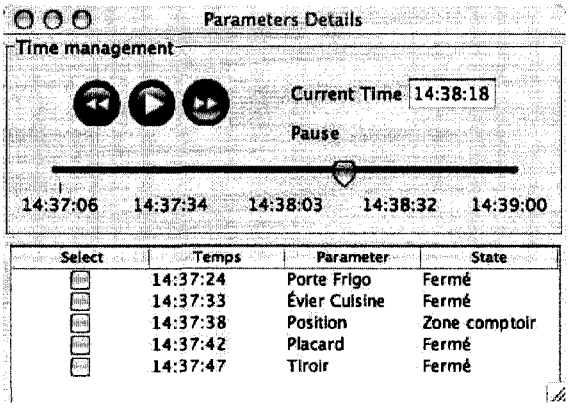

Figure 22 – Vue générale des paramètres

Commentaires sur l'interface de validation

L'interface de validation permet de suivre avec plus de facilité les sorties du simulateur au cours du temps. Grâce aux itérations du cycle de prototypage *lo-fi*, son fonctionnement a été simplifié. Les différentes vues résultantes offrent un suivi des principaux concepts du simulateur d'AVQ ; les activités et leur cycle de vie, les paramètres et leurs états, les ressources et leur utilisation. Pour les futurs utilisateurs du simulateur, l'interface de validation est un outil d'interprétation des données simulées pour s'assurer d'une cohérence avec les données issues du monde réel. Ces utilisateurs peuvent désormais observer une simulation sous divers angles pour valider chaque aspect de leur simulation et vérifier les temps des transitions à l'écran.

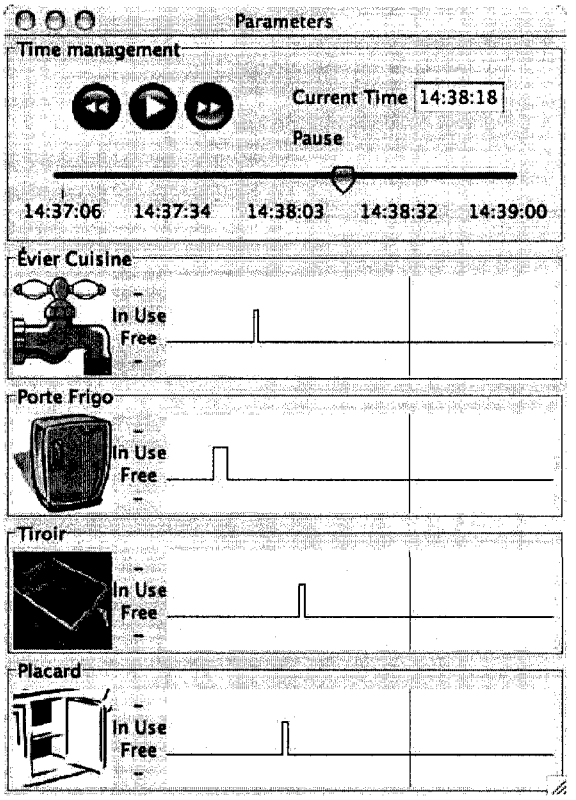

Figure 23 – Vue détaillée des paramètres

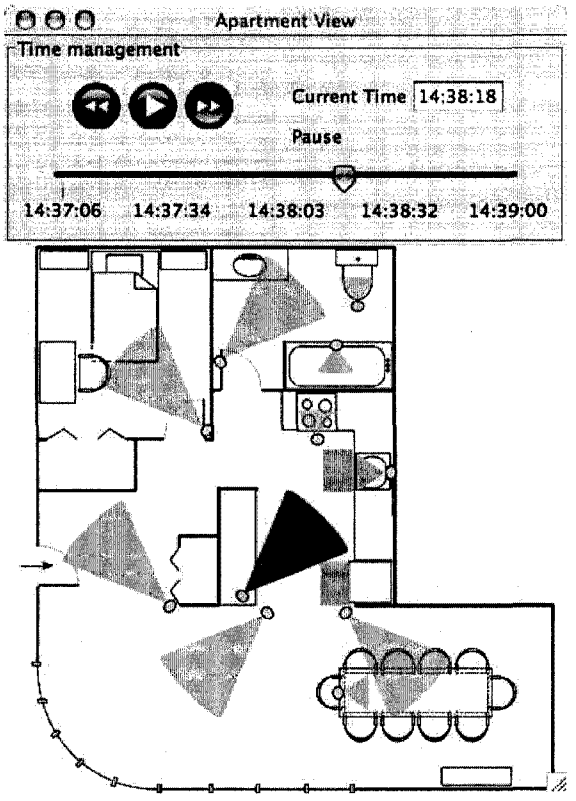

Figure 24 – Plan de l'appartement

Conclusion

Contributions

L'objectif à long terme du simulateur d'AVQ est la génération automatique de sorties capteurs pour alimenter une base de connaissances d'AVQ correspondant au comportement sain d'un individu et faciliter la reconnaissance d'activités lors de l'analyse des sorties capteurs en milieu réel.

Le simulateur a fait l'objet d'une précédente maîtrise par Abdelkader Ghouraf. Son travail a permis d'identifier les fondements théoriques du simulation d'AVQ et d'aboutir à une implémentation initiale du simulateur.

L'objectif général de ce mémoire est de valider les résultats issus de ce simulateur. Cependant, une refactorisation du code était nécessaire avant de continuer le développement du simulateur. Le travail s'est donc déroulé en deux étapes ; une étape de refactorisation pour faciliter la maintenance et ajouter des nouvelles fonctionnalités au simulateur, et une étape de validation.

La refactorisation a demandé de suivre les normes de programmation de développement Java, d'utiliser des *design patterns* autant que possible et de restructurer les objets Java pour refléter leur rôle. Grâce aux techniques de refactorisation existante, le simulateur d'AVQ a subi un profond changement qui le rend plus facile à utiliser et à maintenir.

L'étape de validation a nécessité plus de travail de recherche pour trouver les différentes techniques de validations existantes et déterminer les plus éloquentes pour le simulateur.

Parmi les techniques trouvées, deux d'entre elles ont été implémentées : *historical data validation* et *validation par animation.* Pour la première technique, des sorties capteurs ont été enregistrées lors d'une expérimentation en laboratoire pour être comparées avec celles issues du simulateur pour les mêmes activités. Cette comparaison est subjective puisqu'aucun critère n'a besoin d'être défini au préalable, mais permet d'assurer une première correspondance entre les sorties du simulateur et le monde réel. Ce travail a fait l'objet d'un article présenté en 2006 dans la conférence *Summer Computer Simulation Conference.*

La *validation par animation* offre une interface pour visionner graphiquement la simulation. Cet outil a fait l'objet de deux projets de baccalauréat pour définir au mieux l'interface au complet avant son implémentation. Puis nous l'avons implémenté et testé avec les données du simulateur relatives à l'expérience en laboratoire. Suite à une simulation, l'utilisateur peut désormais charger ses données dans l'interface plutôt que de lire des fichiers XML manuellement.

Critiques du travail

Comme établis dans le chapitre *Objectifs et Méthodologie*, le travail effectué suivait deux axes, la refactorisation du code et la validation du simulateur.

Bien qu'il soit difficile, voire impossible de déterminer la fin de la refactorisation, nous pouvons évaluer ses bénéfices en comparant le code original et le code actuel. La refactorisation a cessé lorsque les objets Java et les *packages* étaient suffisamment représentatifs de leurs fonctions et lorsque toutes les fonctionnalités (interruptions, hiérarchies des activités, etc.) du simulateur ont été implémentées. Le code est maintenant plus compréhensible et son évolution sera plus facile à mettre en oeuvre pour les prochains développeurs.

L'importance de la refactorisation apparaît jusque dans l'implémentation de l'interface graphique. Le développement a été échelonné sur plusieurs mois, souvent entrecoupés

pour réaliser d'autres tâches. Chaque reprise de la programmation permettait de réviser le code pour faciliter la programmation des interfaces suivantes et garder un code homogène.

La validation est beaucoup plus difficile à évaluer, et d'autant plus subjective qu'aucun critère d'évaluation n'est préconisé par les techniques de validation, mis à part le bon sens. Pour la première technique basée sur la comparaison entre des données réelles et d'autres issues du simulateur, l'ordre et le temps des changements d'états des paramètres sont comparés manuellement pour vérifier leur crédibilité et leur correspondance avec la réalité. Ces comparaisons sont bonnes puisque l'on remarque les correspondances entre les activités observées sur la vidéo et les transitions des activités, des ressources et des paramètres obtenues par le simulateur.

La technique de validation par animation offre, pour sa part, un bon outil d'interprétation des résultats. L'utilisateur du simulateur peut désormais suivre le déroulement de sa simulation sur plusieurs vues dépendamment des concepts à observer. Il est alors aisé de comparer la vidéo de l'expérience avec les sorties du simulateur en utilisant l'interface de validation.

Travaux futurs de recherche

Le simulateur d'AVQ nécessite encore de la recherche et du développement pour atteindre l'objectif de génération automatique de sorties capteurs. Parmi les prochaines étapes, rendre les sorties capteurs aléatoires et créer une interface de gestion de scénarios et de gestion de l'environnement sont importantes. Tout d'abord, l'aspect aléatoire sera essentiel lorsqu'il faudra générer une quantité massive de données. Cela pourra se faire en décalant l'horaire de départ des activités, en le propageant sur les sous-activités et en modifiant le temps d'exécution des tâches et sous-tâches. Certaines activités peuvent

également être exécutées dans des ordres différents. L'altération de ces différents paramètres devraient permettre l'élaboration d'une première version aléatoire du simulateur. La seconde étape, créer une interface de gestion des scénarios et de l'environnement, devrait faciliter la création de scénarios, d'AVQ et de dépendances entre les ressources, les paramètres et les activités. La troisième étape consiste à établir un critère d'équivalence entre les scénarios pour s'assurer que les scénarios simulés sont équivalents à des scénarios réels. Enfin, la dernière étape consiste à implémenter les fenêtres de chargement des scénarios et de gestion du temps, comme indiqué lors du prototypage *lo-fi* pour observer simultanément un ou plusieurs scénarios et synchroniser une sélection de vue dans le temps.

Annexe A

Extraits du prototype Lo-Fi

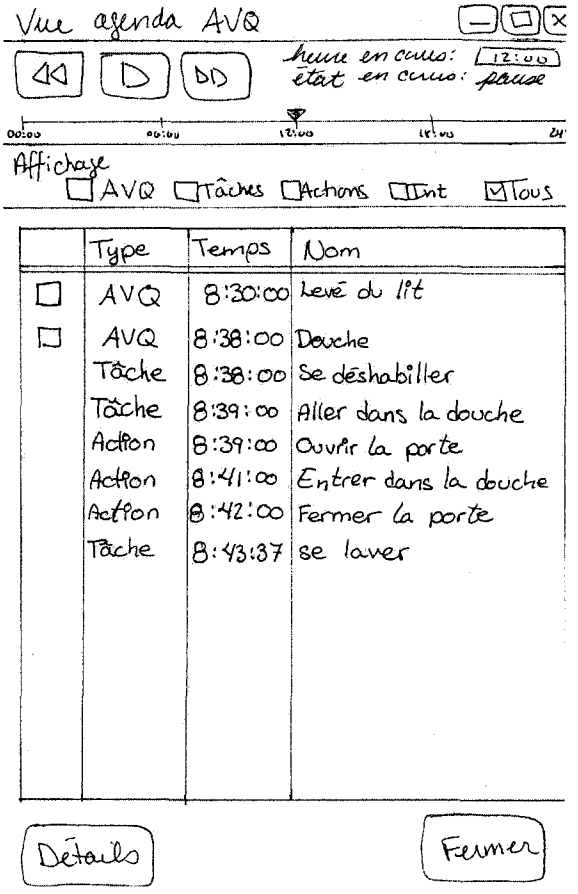

Figure 25 – Vue générale des AVQ

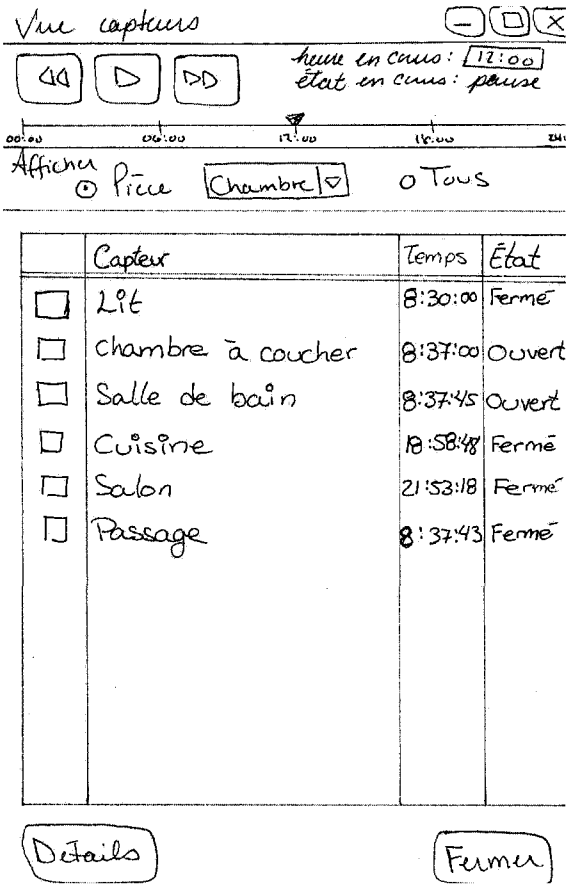

Figure 26 – Vue générale des paramètres

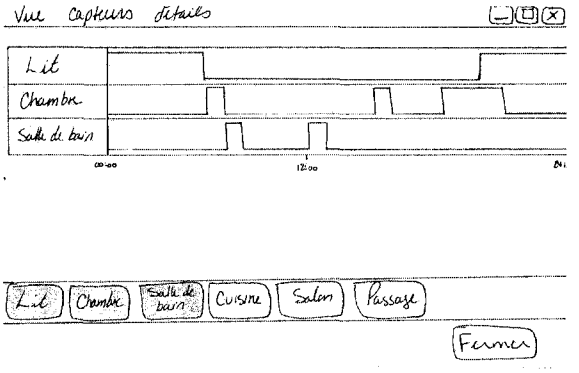

Figure 27 – Vue détaillée des paramètres

Annexe B

Article de conférence *Validation process of an activity of daily living simulator*

Article de conférence internationale à comité de lecture. Paru dans *Proceedings of the 2006 Summer Computer Simulation Conference (SCSC 06). 2006, pages 356 à 362*, en collaboration avec Abdelkader Ghouraf et Hélène Pigot.

Validation Process of an Activity of Daily Living Simulator

Pierre Busnel, Abdelkader Ghouraf, Hélène Pigot
Laboratoire Domus
Université de Sherbrooke, QC, Canada
{Pierre.Busnel, Abdelkader.Ghouraf, Helene.Pigot}@USherbrooke.ca

ABSTRACT

This paper presents the development of a simulator of Activities of Daily Living in the context of smart homes. In such environment, activity recognition requires a large amount of data collected by sensors dispatched in the house during several years of experimentation. For a matter of time and convenience, activities in a smart home are simulated. Based on theoretical models upon the human behaviors, the simulator is implemented with object oriented Petri Networks. The activity of daily living simulator is under validation process. The validation by animation is the first to be applied.

KEYWORDS

Petri Networks, smart home, validation, activity of daily living, cognitive assistance

1 SMART HOME AND COGNITIVE ASSISTANCE

The occidental countries face an increasing of elders joint to a decreasing birth rate. According to U.S. Census Bureau estimations, the population aged 65 and older represented 36 million persons in 2003 and is projected to increase to 72 million in 2030 and 86.7 million in 2050 [He 2005]. This situation increases the health injuries due to normal and pathological aging which head toward loss of autonomy and fragility reducing the quality of life of the aged.

Alzheimer disease is one of the most known disease related to aging. Based on the previous census, recent estimations indicate that 4.5 millions americans were suffering from Alzheimer disease in 2000 rising to 13.2 million persons by 2050 [Hebert 2003].

The Alzheimer disease is an irreversible stepping disease which causes mainly cognitive losses [Reisberg 1982]. Deficits in memory, planning and judgement cause a disruption of autonomy. In the first steps of the disease, people encounters difficulties to perform unusual or complex tasks but remains able to stay at home in an habitual setting. Risks of isolation, malnutrition, fire-burns, water-flow, and fire incidents arise as the disease evolves. In the late steps of the disease, people needs a continuous supervision, which necessitates more often a transfer to a hospital setting.

Despite of the impairments, 92% of the elders would like to stay at home and 40% would rather spend the end of their life at home. Therefore, they rely upon their family and/or caregivers to help them in their everyday life. Fulfill all the elders needs become rapidly exhausting for the caregivers. To alleviate the caregivers burden, an assistive environment could enhance the autonomy of the elder by providing appropriate cues in order to perform safely the activities of daily living (ADL).

Thanks to ubiquitous and pervasive computing, a smart home interacts with the resident to foster autonomy [Jorge 2001]. In opposition to tele-monitoring, where help is provided from an external source, smart homes are similar to a closed system composed of the home and the resident. The smart home is usually equipped with various sensors connected to a server that gathers and analyses the data thanks to artificial intelligence technics. Depending on the diagnosis upon the performance, the smart home intervenes to avoid situations risks and help people to complete activities.

Assistance is divided into two types: the physical assistance and the cognitive one. In the first

case, the environment aims to compensate the physical handicaps by providing appropriate tools such as remote controls, voice recognition, wheelchair ... In the case of cognitive assistance, the environment aims to compensate the cognitive impairments by prompting advices and interacting with the individual according to the situation.

At the Domus laboratory, the cognitive assistance is oriented towards people suffering from cognitive impairments [Pigot 2003]. The concept of assistive devices provide multiple challenges in ubiquitous computing, context awareness, artificial intelligence, networking and interface conception [Jorge 2001].

In the Domus laboratory, an apartment is equipped of electro-magnetic sensors to detect the opening of a door, sensitive rugs and motion sensors to locate the resident. The activity performed in the apartment is recognized by the sensors data analysis. In daily life, people performs activities in various manner. For example, in the morning routine, one could awake at different time, decides to get to toilet or else to go eating breakfast first. He also could be interrupted by a phone call or stay longer in the kitchen to listen to radio. So, even for the same activity the sensors could be triggered in a different sequence and at a different time and duration. To address recognition issues, data from various scenarios of the same activity should be gathered in a database to provide regular and abnormal behaviors schemes. Unfortunately it would take months or even years to obtain such data. A simulator of ADLs is then necessary to generate resident behaviors similar to the everyday life where ADL should be interrupted because of phone call or performed on a longer time because of tiredness.

In the last decades, research has been done to find protocols to model complex systems rapidly and efficiently. The aim is to develop simulators able to produce similar results as those obtained in the real world but faster and in a larger amount. Therefore, simulation models have to be the more accurate as possible. It implies deep analysis of the real world during the simulator development. The conception of a simulator starts with a description phase of the experiment objectives where the problem is defined and real world investigated to propose system theories. Then, a conceptual model is formulated by modeling the previous theories leading to the implementation of the simulation model. The second phase is iterative to refine models [Balci 1994, Nance 1983, Sargent 2004]. Both methods integrate validation and verification at various steps of the development, but Sargent's paradigm includes a closer validation between system results and simulation model results (Figure 1). It is then privileged in the approach of the ADL simulator.

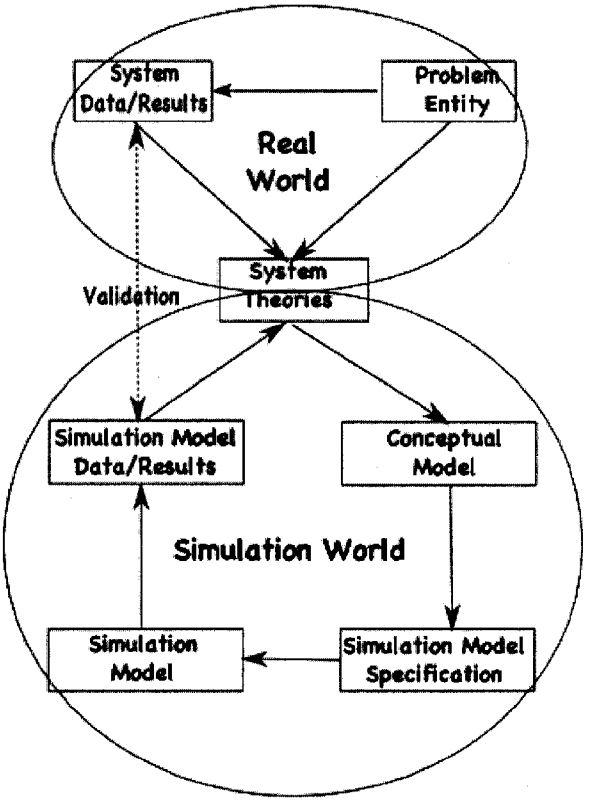

Figure 1 : Real World and Simulation World Relationship adapted from Sargent [2004]

2 REAL WORLD

The real world is composed of an individual performing ADLs at home. All the day, the individual prepares lunch, goes to the bathroom to wash himself, watches TV or reads journals and goes to bed. Sometimes he performs the ADL without interruptions, sometimes he performs several ADLs in parallel or is interrupted by prior situations. For instance, he answers to phone call while cooking pasta. The 24 hours day are divided into life habits where some ADLs are more likely to occur. In a smart home, the sensors dispatched into the environment collect data upon the activities performed. The system theories describe the

following characteristics of the real world : the ADLs and life habits, the management of interruptions, the house map and its sensors, the individual abilities and the available resources.

2.1 ADLs and life habits

An ADL is an activity performed to fulfill the basic needs of an individual in order to take care of himself and his house [Carswell 2004]. The ADL is decomposed hierarchically in tasks and subtasks until reaching the atomic actions perceptible by sensors. For instance, the ADL *making tea* is composed of tasks such as *boiling water* composed of actions such as *opening the tap*. An action modifies the status of the environment and the person. For instance, after having drunk water, the person is no more thirsty and the glass is empty. The actions modify also the status of the task and the ADL as described below in the activity life cycle.

The life habits refer to routines which are performed on a periodical base; one drinks coffee 6 days a week for breakfast between 7:30 am and 8:00 am and eat eggs and bacon on sunday morning. The life habits play a significant role to predict and recognize more accurately the ADLs performed.

2.2 Interruptions

As one can't foresee when the phone is going to ring or a friend to knock at the door, he has to adapt the current ADL to unpredictable situations. In the real world, an activity interrupted should be resumed later or started over again or even postponed. For instance, when *cooking pasta*, while the water is boiling, one could be interrupted by a phone call and safely turns off the stove before answering. Depending of the duration of the phone call, the water is still warm or need to be reheated. In the first case, we can resume the task, otherwise we do have to start boiling water from the beginning. The validity time indicates the need to resume or restart the task.

2.3 Activity life cycle

ADLs and tasks evolve accordingly to a activity life cycle, which defines 9 distinct states. The *released* state characterizes all the ADLs and tasks, that are not planned in the next period. It could be because they are not planned or potentially occurring in an interruption, it could be also because they have been previously *completed.*

The initial state is *scheduled* when the activity is planned in the period or *unpredictable* if the activity is specified as an interruption. When all the resources are available, the activity state becomes *waiting.* Then, the activity status is changed to *execution* if all the pre-conditions are fulfilled without exceeding the waiting time. Otherwise, the activity is *cancelled* and *released.* When executed in the ideal situation, the activity is then *completed* and *released.* However, as seen in the previous paragraph, activities may be *interrupted* before their completion. Therefore, the partial results of the activity depend on the validity time of the activity. The validity time determines the delay during which the partial results remain valid. The interruption time characterizes the period of time during which the activity is *interrupted.* If the interruption time doesn't exceed the validity time, the partial results remain valid and the activity can be resumed. Otherwise, the partial results are invalid and the activity is *abandoned* then *released.* Once *released,* the activity may be *scheduled* again.

2.4 Resources

The resources are defined by all the items needed to perform a task either from the environment or related to the individual. They may be limited in the time. They are essential for the accomplishment of a task. The human resources, according to the VACP theory, are described by four processing components [Wickens 84]. Standing for Visual Audio Cognitive and Psychomotor components, they represent respectively the capacity to see, to hear, to remember procedural and declarative knowledge and, last, the ability to accomplish movements.

The environmental resources are divided into consumable and material resources. The first ones regroups all the substances or products needed in the daily life (water, milk, soap ...). The material resources are composed of all the equipments of the apartment; from the appliances such as the fridge to the pieces of furniture such as the chairs or the plates. These resources support the task. Some of them could be used for more than one task at a time. For instance, a stove is equipped with an oven and four cook-tops that can be used simultaneously for the tasks *cooking eggs* and *boiling water.*

2.5 Environment

The environment is defined by the rooms configuration and the sensors dispatched. The rooms configuration is essential to estimate the paths into the environment. The sensors give informations

about the individual physiological state, his location or his actions. Bistable and analog sensors are used. A bistable sensor is a sensor that has two resting states, both of them are stable. The signal sent indicates the current states of the sensor. For instance, sensitive rugs, electro-magnetic sensors, motion sensors are bistable. An analog sensor sends data representing a measurable value. For instance, flow-meters, thermometers are analog.

3 CONCEPTUAL MODEL

The conceptual model should respect the problem entity and express the above system theories. The activities performed in a smart home are like processes running on a computer; both need limited resources and both could be done in parallel or be interrupted. The analogy between the task executions in a smart home and the computer processes leads to the use of Petri Networks to simulate ADLs as they have been extensively used to simulate computer processes [Peterson 77].

The Petri networks describe the activity sequences, the resources and the potential interruptions. Their main interest lies in the power of expressivity they offer thanks to the graphical language and the formal representation. The dynamic characteristics of the systems are displayed on the Petri Networks in order to understand its behaviour and to study its performance.

The Petri Networks are composed of places, transitions and arcs. An input arc connects a place to a transition and an output arc connects a transition to a place. A place may contain consumable tokens. When the places connected to a transition contain the number of tokens required to cross a transition, the transition fires, consumes the tokens from the input places and adds some tokens in the output places. The number of tokens added in the output places is not necessarily equal to the number of tokens consumed and depends on the transition. The current state of the system is given by the number of tokens dispatched in each place of the Petri Network.

In the ADL simulator, the places represent the entering conditions to fulfill in order to cross a transition or the outgoing conditions that are fulfilled after a transition fires. A transition is linked to an activity. When it fires, the activity is under realization.

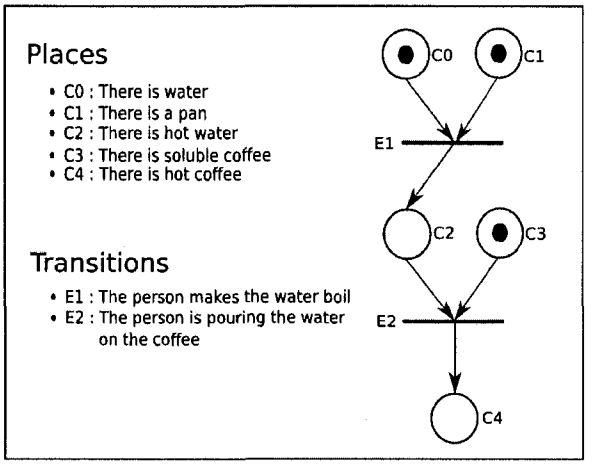

Figure 2 : Petri Network modeling the activity of making instant coffee

The figure 2 illustrates the activity of making instant coffee, the five places (C0 to C4) represent the conditions related to the environment, such as *having water* and *having a pan.* The tokens in the places C0, C1 and C3 symbolize the fulfilled associated conditions. When tokens are in the places C0 and C1, the entering conditions are fulfilled and the transition fires the event E1. In the next state of the Petri Network, the places C0 and C1 are empty and the place C2 contains a token. Unless we ran out of soluble coffee, the place C3 always contains a token.

The example of figure 2 presents an activity which has been completed without interruption. The activity life cycle described above leads to 4 different paths for a task through the 9 existing states. These 4 paths, which are part of the global Petri Network, are the following ones :

- the task is completed without interruption,
- the task constraints are never satisfied, then the task is cancelled after the waiting period is over,
- the task is interrupted for a short time then is resumed until completion,
- the task is interrupted for a long time then it is either reseted or abandoned.

4 IMPLEMENTATION

The ADL simulator generates an agenda of the ADLs performed at a precise time and place. The figure 3 illustrates the simulator inputs and outputs.

The user determines a global scenario as input that regroups :

- an ADLs scenario,
- the individual life habits,
- the unpredictable activities,
- the resources related to the individual,
- the environment resources.

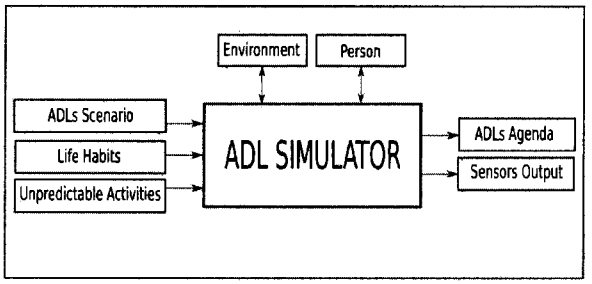

Figure 3 : ADL simulator inputs and outputs

ADLs scenario contains all the ADLs to be performed, their decomposition in tasks and actions, their validity time and execution time during a time period. The same ADL may be described by different sequences of tasks and actions. Life habits are defined by the frequency and the period when the ADLs are usually performed for an individual. Unpredictable activities are the interruptions the user wishes to occur during a simulation period. They may occur or not, and depending on the current ADL in execution, they may be ignored or performed. If the interruption is performed, the current ADL is interrupted as described in the activity life cycle.

The output ADLs agenda, describes all the ADLs, tasks and actions that occurred at a determined time. It also contains all the life cycle states of an activity and the time the states changed. This includes the interruption wether they were performed or ignored.

The figure 4.a illustrates a simplified ADLs scenario and the life habits related. It does not include the decomposition in tasks and actions. The simulation period is from 7:30 to 8:00 am. There is two ADLs, *washing hands* and *washing face* that usually take 40 seconds and 100 seconds respectively to perform. The individual is used to perform them between 7:40 and 8:00 am. The figure 4.b shows the unpredictable activities the user wished to occur. The phone is to be ringing between 7:40 and 7:50 am.

The ADLs agenda on figure 5 is the list of all the activities transitions. The ADL *washing hands* started at 7:42 and lasted 42 seconds. Then, the ADL *washing face* started until the end of the simulation, 100 seconds later. A phone call occurred at 7:43 but was ignored.

ADLs Scenario		**Life Habits**
Simulation Period 7:30 am to 8:00 am		
Activity	**Duration**	**Period**
Wash hands	40 s	7:40 am to 8:00 am
Wash face	100 s	7:40 am to 8:00 am

Figure 4.a : ADLs scenario and Life Habits

Unpredictable Activities		
Activity	**Duration**	**Period**
Phone call	35 s	7:40 am to 7:50 am

Figure 4.b : Unpredictable activities

ADLs Agenda		
Activity	**Transitions**	**Time**
Wash Hands	*Scheduled* to *Waiting*	7:42:00
Wash Hands	*Waiting* to *Execution*	7:42:00
Wash Face	*Scheduled* to *Waiting*	7:42:00
Wash Hands	*Execution* to *Completed*	7:42:42
Wash Face	*Waiting* to *Execution*	7:42:42
Phone Call	*Unscheduled* to *Waiting*	7:43:00
Phone Call	*Waiting* to *Cancelled*	7:43:00
Wash Face	*Execution* to *Completed*	7:44:22

Figure 5 : ADLs agenda

In the ADL simulator, object oriented Petri Networks are used to enhance the resources expressivity. Object oriented Petri Networks add properties to tokens which evolve through time or when a transition fires [Sibertin-Blanc 1997]. These

tokens are objects and can be affected when an ADL is performed. For instance, the token representing the soap decreases the quantity left each time the activity *washing hands* is performed. The environmental and personal resources are represented in the simulator by these tokens. An ADL consumes or modifies resources from the environment and related to the individual, therefore the simulator interacts with both of them.

5 VALIDATION

In this section, we discuss model validity and validation techniques. Numerous techniques are proposed to validate the accuracy of a model but no universal technique will conclude in the failure or in the success of a model. It is recommended to use a combination of several validation techniques to increase the model validity [Sargent 04]. The following sections will describe various validation techniques. We will retain for the beginning the animation technique.

5.1 Validation Techniques

The 3 following validation techniques have been chosen for their abilities to validate specific properties if the ADL Simulator. These techniques are event validity, historical data validation and animation.

5.1.1 Event validity

According to Sargent [2004], the occurrences of the simulation model are compared to those of the real system to determine if they are similar. In the ADL simulator, to satisfy the event validity, the ADLs agenda must be similar to the life habits. It needs to generate simulated data for a long period. For instance, one's life habit is to drink coffee 5 times a week and tea twice a week. To validate this habit, simulated data will be generated for a year. The outputs tendency should reflect the life habits of making coffee five times a week and making tea twice a week.

5.1.2 Historical data validation

Historical data validation consists of a comparison between data collected in the real world and data obtained through the model to determine the realistic model behavior. For instance, data such as sensors outputs are collected in the environment while a user is performing ADLs. The same ADLs are described in a scenario submitted to the ADL generator. It generates simulated sensors outputs which are compared to the ones collected in the apartment. It supposes to establish a distance to compare the ADL simulated and the ADL performed in the real world.

5.1.3 Animation

A graphical display of the model behavior is implemented to see how it evolves in real time. For instance, a map can display the room where ADLs are performed, highlight activated sensors and display analog sensors values. Such animation is useful to check consistence upon the simulated data.

5.2 Animation validation technique

The animation technique of the ADL simulator consists in displaying graphically the behavior of the model during the simulation of an ADLs series. By implementing interfaces for specific views, the user is able to oversee the progress of all the aspects of the simulations.

These interfaces represent the following views: the map view, the ADLs agenda view, the sensors agenda view, the personal resources view, the environmental resources view and the time view.

The map view displays a global view of the environment. Motion sensors, sensitive rugs and magnetic sensors attached to doors are represented on a map dispatched in the rooms of the apartment. This view is convenient to locate places where ADLs are performed.

The ADLs agenda view displays a list of all the ADLs, tasks and actions in progress or done of the current scenario. The user can choose wether to display ADLs, tasks or actions. Activities execution time and description are also displayed.

The sensors agenda view shows the list of the sensors that are activated or that have been activated already with their states and their description.

The resources view displays the list of the resources available associated to their status and the quantity left. The resources related to the patient are similarly displayed in another view.

The 5 previous views also give access to detail views. The map details view consists in a closer view of the selected room or specific angle, the ADLs agenda details view shows the hierarchical composition of the ADLs tasks, subtasks and actions as a tree using different colors to set apart activities that are done, in progress or yet to do. The user can follow the sequences of tasks and actions in the current ADL or even in concurrent ADLs. The

sensor and resource detail views show status changes on a time scale.

The simulation is controlled through a time panel which allows the user to play, pause, fast forward or backward the scenario evolution.

The interface enhances the possibilities to take a close look at the simulation, the user can check the location, resources or sensors status at a given time or on a specific time period. Thanks to the different views provided, one could compare different ADLs realizations, check the resources evolution related to the patient or from the environment and verify the consistence of an agenda.

6 CONCLUSION

The ADL simulator development and validation are complex processes that require a deep comprehension of the real world problem entity. The model theories used in the simulator are based on VACP theory, activity theory, occupational therapy theory.

Petri Networks are widely used in computer simulation and are convenient to represent chronological events. The object oriented Petri Network are needed to express the quantity of resources available. The ADLs are simulated with object oriented Petri Networks, which represents the activity life cycle taking in account the occurrence of interruption during activity execution.

The first validation technic to be implemented is the validation by animation. Different views of the inputs and outputs simulator are generated allowing the user to follow the evolution of data during time and to compose different occurrences of the same activity. Validations by event validity and historical data will then be implemented later.

REFERENCES

Balci O. 1994. "Validation, verification, and testing techniques throughout the life cycle of a simulation study.", In *Proceedings of the 1994 Winter Simulation Conference*, 215-220

Carswell A. et al. 2004. "The Canadian Occupational Performance Measure: A research and clinical literature review", In Canadian Journal of occupational therapy, 71(4), 210-222

He W. et al. 2005. "65+ in the U.S.: Current Population Reports" Washington, DC: U.S. Bureau of the Census, 23-209

Hebert L. et al. 2003. "Alzheimer Disease in the US population", *Prevalence Estimates Using the 2000 census*, Arch Neurol, Vol. 60

Jorge J. 2001. "Adaptive Tools for the Elderly New Devices to cope with Age-Induced Cognitive Disabilities", In *Proceedings of the 2001 WUAUC*, 66-70

Nance R. 1983. "A tutorial view of simulation model development", In *Proceeding of the 1983 Winter Simulation Conference*, 16-22

Peterson J. L. 1997. "Petri Nets", Computing Surveys, Vol 9, No. 3

Pigot H., et al. 2003. "The intelligent habitat and everyday life activity support", In *Proceedings of the 5th international conference on Simulations in Biomedicine*, Slovenia

Reisberg B. et al. 1982. "Global Deterioration Scale", American Journal of Psychiatry, 139, 1136-1139

Sargent R. 2004. "Validation and verification of simulation models", In *Proceeding of the 2004 Winter Simulation Conference*, 17-28

Sibertin-Blanc C. 1997. "Concurrency in CoOperative Objects", In *Proceedings of the Second International Workshop on High-Level Parallel Programming Models and Supportive Environments*, HIPS'97, Geneva, 35-36

Wickens C., *Engineering psychology and human performance*, Glenview, IL: Scott, Foresman and Company, 138-145, 1984

Bibliographie

[1] O. Balci. Validation, verification, and testing techniques throughout the life cycle of a simulation study. In *Proceedings of the 1994 Winter Simulation Conference*, pages 215–220, 1994.

[2] P. Busnel, A. Ghouraf, and H. Pigot. Validation process of an activity of daily living simulator. In *Proceedings of the 2006 Summer Computer Simulation Conference (SCSC 06)*, pages 356–362, 2006.

[3] A. Carswell et al. The canadian occupational performance measure : A research and clinical literature review. *Canadian Journal of Occupational Therapy*, pages 210–222, 2004.

[4] B. Reisberg et al. Global deterioration scale. *American Journal of Psychiatry*, 139 :1136–1139, 1982.

[5] H. Pigot et al. The intelligent habitat and everyday life activity support. *Proceedings of the 5th International Conference on Simulations in Biomedicine*, 2003.

[6] L. Hebert et al. Alzheimer disease in the US population. *Prevalence Estimates Using the 2000 Census*, 60, 2003.

[7] W. He et al. 65+ in the U.S. : Current population reports. Technical report, Washington, DC : U.S. Bureau of the Census, 2005.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring, Improving the Design of Existing Code.* Addison-Wesley, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[10] A. Ghouraf. *Simulation des activités de la vie quotidienne dans un habitat intelligent par des réseaux de Petri.* Mémoire de maîtrise, Université de Sherbrooke, 2005.

[11] P. Jaulent. *Génie Logiciel, les méthodes.* Armand Colin, 1992.

[12] J. Jorge. Adaptive tools for the elderly new devices to cope with age-induced cognitive disabilities. In *Proceedings of the 2001 WUAUC*, pages 66–70, 2001.

[13] R. Nance. A tutorial view of simulation model development. In *Proceedings of the 1983 Winter Simulation Conference*, pages 16–22, 1983.

[14] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3) :223–252, 1977.

[15] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[16] M. Rettig. Prototyping for tiny fingers. *Communications of the ACM*, 37(4) :21–27, 1994.

[17] V. Rialle, C. Ollivet, P. Rumeau, A. Serna, H. Pigot, and C. Hervé. Ethique des technologies émergentes pour l'aide aux malades "Alzheimer" et à leurs aidants. In *VIIIe Congrès International Francophone De Gérontologie Et Gériatrie*, 2006.

[18] R. Sargent. Validation and verification of simulation models. *Proceedings of the 2004 Winter Simulation Conference*, pages 17–28, 2004.

[19] C. Wickens. *Engineering Psychology and Human Performance.* Scott, Foresman and Company, 1984.